Programming distributed and adaptable autonomous components–the GCM/ProActive framework

Françoise Baude, Ludovic Henrio, Cristian Ruz

SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2013; **00**:1–38
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Programming Distributed and Adaptable Autonomous Components – the GCM/ProActive Framework

Françoise Baude[1], Ludovic Henrio[1], Cristian Ruz[2*]

[1] *Univ. Nice Sophia Antipolis, CNRS, I3S, UMR 7271,*
INRIA 2004 Route de Lucioles, BP 93, 06902 Sophia Antipolis CEDEX, France
[2] *Department of Computer Science, Pontificia Universidad Católica de Chile,*
Av. Vicuña Mackenna 4860, Macul, Santiago, Chile

SUMMARY

Component oriented software has become a useful tool to build larger and more complex systems by describing the application in terms of encapsulated, loosely-coupled entities called components. At the same time, asynchronous programming patterns allow for the development of efficient distributed applications. While several component models and frameworks have been proposed, most of them tightly integrate the component model with the middleware they run upon. This intertwining is generally implicit and not discussed, leading to entangled, hard to maintain code. This article describes our efforts in the development of the GCM/ProActive framework for providing distributed and adaptable autonomous components. GCM/ProActive integrates a component model designed for execution on large scale environments, with a programming model based on active objects allowing a high degree of distribution and concurrency. This new integrated model provides a more powerful development, composition, and execution environment than other distributed component frameworks. We illustrate that GCM/ProActive is particularly adapted to the programming of autonomic component systems, and to the integration into a service oriented environment.

Received . . .

KEY WORDS: component based software engineering; reconfiguration; distribution; autonomicity

1. INTRODUCTION

Software components have been designed to provide composition frameworks raising the level of abstraction compared to objects. Components split the application programming into two phases: the writing of basic business code, and the composition phase consisting in plugging together the basic blocks programmed above. To scale better and ease the programming of large applications, *hierarchical* component models allow each component to be composed of other components. In the context of the CoreGrid European Network of Excellence, a component model called GCM [1] (Grid Component Model) has been designed; GCM is an adaptation of Fractal [2] for large scale distributed computing. It defines structurally the way to compose components. The GCM component model, its API, ADL and its deployment capacities has been standardised by ETSI [3, 4, 5, 6]. However, getting a component-oriented programming model not only requires a specification, but also a well-thought implementation: components behaviour depends of their underlying semantics resulting from the chosen implementation technology. This paper adds an original piece of work around the GCM efforts by diving into that aspect given a specific implementation technology.

*Correspondence to: cruz@ing.puc.cl

Indeed, one of the original aspects of this paper is that it focuses on the interplay between the component model and the underlying programming model. The distributed programming model we rely on is based on the notion of active objects. Active object [7] is a programming pattern adapted to distributed computing. The principle of active objects is very simple: an object is said to be active if it has its own thread. Every call to an active object will be some form of remote method invocation; we call such an invocation a *request*. An active object handles invocations asynchronously: when a request is received it is put in a queue, and will be served later by the active object. One crucial point of active objects is that they are mono-threaded and there is no shared memory between two active objects. Consequently, active objects prevent data race-conditions without having to use locks or synchronised blocks. The only race-condition that exists in this programming model is the concurrent request invocations toward the same object; this is the only concurrent aspect that exist in the model. An active object can act as an access point to a set of objects but two different active objects cannot share memory. As a consequence, this programming model is particularly adapted to develop distributed applications because each active object can be deployed at a different location and is by construction loosely coupled with the other active objects.

We identify three main contributions for this paper.

Contribution 1. *Autonomous Components: A Programming Model for Distributed Components*

Most research contributions on component models focus on the component structure [2, 8, 9], its control and evolution [10, 11, 12], or the specification of adequate coordination/communication modes between components, e.g. parallel communications between components [13, 14], orchestration of services exposed by components for SOA (Service Oriented Architecture) [15]. Other works in the component community aim to dig into efficient ways to implement the component model in specific contexts leveraging well chosen supporting environment or middleware for deploying and running the components. These later approaches are bottom-up: first, authors select the middleware they want to rest upon, then they design a software component model in which they introduce features and related implementation in a way that efficiently uses the desirable features from the underlying middleware. Some of those component models have been proposed for distributed application programming and execution, like for example: CCA and Assist component models on top of web service based grid standards [16, 17], PaCO and GridCCM on top of Corba object request brokers [18], CCA on top of H2O [19]. The drawback of this approach is that the features of the underlying middleware impact too much the resulting component model, its implementation, and its runtime behaviour, leading to entangled, hard to maintain code. We advocate another, more self-contained, approach. Given the features we believe are needed at the component model level, we carefully devise a component model implementation independently of the underlying middleware selected to effectively deploy and run the application. Instead, we underline the relation between the component model and its programming and execution model, i.e., the behaviour of the component-based application. In other words, we make explicit the resulting runtime behaviour of the component model implementation without having to consider the effective supporting environment. To the best of our knowledge, the only examples of such kinds of discussion relating a component model with the programming model chosen to implement it are around: aspects [20], mixins [21], and direct enhancements of the core programming language as Java [22] to implement component constructions. Consequently, *we are among the first ones to study the relation between a programming model for distributed computing and a component model targeting distributed systems*.

In this paper, our goal is to illustrate the benefit of chosing an adequate programming model, based on our own experience implementing a sophisticated component model. In particular, we rely on a very powerful programming model, the active object model, to implement our component model. Also, contrarily to the research contributions on component models mentioned above, our programming model is a runtime model, expressing how programs run, and not only how to compose pieces of code. The study of the interplay between a component model used to compose applications and a programming model specifying how applications run is the main originality of our approach. We have put a particular effort to integrate our programming model and our

component model in a common structured programming environment: our components are active objects, they communicate by asynchronous requests, and each of them can be placed on a different location. We think that implementing the component model using this programming model provides a good *abstraction for distribution and concurrency*, and we claim here that this new integrated model provides *a more powerful development, composition, and execution environment than other distributed component frameworks*.

GCM/ProActive is the reference implementation of the GCM component model. It is based on the ProActive Java library [23] and relies on the notion of active objects: each component corresponds at runtime to an active object and consequently each component can easily be deployed on a separate JVM and can be migrated. Of course, this implementation relies on design and implementation choices relative to the purely structural definition provided by the definition of GCM. One of the main advantages of using active objects to implement components is their adaptation to distribution. Indeed, active objects communicate by an asynchronous request/reply mechanism and enforce that each execution thread is isolated in a single active object, and thus a single component. Our programming model provides a natural way to write *loosely coupled components* that do not share memory nor control, and can thus be considered as *independent entities*. We call those loosely coupled components *autonomous components*.

This article also highlights the impact of the programming model as it makes it easy to combine components within a *service oriented environment*, and to program *autonomic component systems*.

Contribution 2. *A Programming and Composition Model for Service Oriented Architectures*

Indeed, until only recently, component based software engineering had not addressed a number of modern software development problems such as interoperability and the diversity of platforms and protocols. Service Oriented Architectures (SOA) aim to cope with these limitations. SOA is an architectural style where applications are conceived as a composition of loosely coupled, coarse grained, technology agnostic, remotely accessed, and interoperable services, bearing thus strong similarities with software components. Adapting component models and corresponding platforms in order to support SOA is thus very relevant for modern software. A typical example of such a marriage of reason [24] has been provided by the SCA (Service Component Architecture) [8] specification supported by industry. Successful integration of SOA interactions into component platforms have been recently illustrated by utility interfaces in the version 2 of the SOFA component model [25], and within our own GCM implementation (as detailed in Section 5).

Contribution 3. *Autonomic Support for Autonomous Components*

Using components to compose applications constrains the programmer to identify the dependencies between components statically. While this limits the programming model expressivity, it eases the design of large applicative architectures, as it provides a static view of the code dependencies. This static view is quite often too restrictive, as application structure needs to evolve dynamically, in particular when facing changes in the execution environment. Some component models, including GCM, address these issues of adaptiveness as a separated concern relative to the business logic: the business code of the component does not deal with the structural changes of the component application. Those structural changes, called *reconfiguration* of the component assembly, are programmed separately and considered as non-functional concerns. Both the decision making and the enactment of structural changes are triggered at the non-functional level. This separation of concerns increases the re-usability of the applicative code, and it also eases the programming of large scale highly adaptive applications. Adaptation procedures can even become smart and generic enough to allow the component system to take reconfiguration decisions in an autonomic manner, where each entity is able to adapt itself to changes in the runtime environment, or in the desired service quality.

In our approach, each component is both autonomous and autonomic in the following sense:

- *Each component runs autonomously.* Components are self-contained entities with their own execution thread and service policy. The composition structure is also the execution

structure; it also defines the data distribution and the interacting entities. We will describe the programming and execution model in Section 3.

- *Each component is able to evolve in an autonomic manner.* This is due to the combination of two features we developed in GCM/ProActive. First, each component can locally execute reconfigurations; this will be described in Section 4.1. Second, each component can take adaptation decisions locally, thanks to an implementation of the autonomic MAPE model [26], see Section 4.2. Both features can be combined to design autonomic behaviour in which decisions are taken in one component, and executed distributedly using uniform interfaces. Compared to previous works [20, 27], the combination of a programming model enforcing decoupling of components and of a precisely defined component architecture for autonomic aspects enables the programming of distributed adaptation procedures, and eases the programming of distributed systems.

Those two aspects compose well together: it is simpler to design generic distributed (self-)adaptation procedures between loosely coupled components.

This paper is organized as follows. Section 2 presents a state of the art about the distributed component models, and the related works. Section 3 presents the programming and execution model of GCM/ProActive. Section 4 describes adaptation and dynamic aspects we developed for GCM components. Section 5 describes the implementation of our component platform; it includes deployment aspects and integration with SCA. An autonomic reconfiguration use-case is presented in Section 6. Finally, after a comparison with the closest related works in Section 7, we conclude this article in Section 8.

2. STATE OF THE ART: DISTRIBUTED COMPONENT MODELS

This section reviews the crucial characteristics that should be featured by a component model in order to support both distribution and autonomic behaviour. For this we will first focus on the aspects of the component models related to the description and evolution of components, especially relatively to autonomic adaptation. In a second step we will present more details about how those basic characteristics relate to distribution management and what features are provided by existing distributed component models, with a particular focus on service-oriented components. This section will conclude by a discussion on the programming models for programming components as autonomous entities, making them better adapted to distribution, easier to program and compose, and easier to control in an autonomic manner.

2.1. General concepts

Component models [28] aim to increase the code reusability, ease of programming, and adaptability of software. They introduce well delimited software entities in the form of components with clearly defined interfaces corresponding to either the offered services or those explicitly required to fulfill them. Component based applications differentiate from other kind of applications (e.g. functional, procedural, object-oriented) by making the resulting software architecture explicit.

2.1.1. Architectural description In general, it is convenient to describe the initial component architecture using an assembly language, also named an Architecture Description Language (ADL), usually an XML dialect. Sometimes, the ADL includes a domain specific language that allows the description of the complete initial architecture by intention in contrast to by extension (concept coined as parameterized ADL, as in Darwin [29]). Associated to an ADL, a factory (generally combining a parser and component generators) instantiates all the components that constitute the application. This step deploys base components from their source implementation and binds component instances according to the dependencies defined in the ADL.

A crucial aspect of architectural component description is the hierarchical composition. A component model is said to be *hierarchical* if components can be glued together to form another

component that itself can be used in the composition; this is, for example, the case of SOFA [10] and Fractal [21]. The composition hierarchy is then a tree with "composite" components as intermediate nodes, and basic blocks containing the real business code as leaves. From a designer point of view, hierarchy eases the creation of large applications by composition of hierarchically built sub-components. The hierarchy underlying the software component architecture is reflected within the ADL associated representation.

2.1.2. Dynamic reconfiguration Some component models allow the modification of the architecture at runtime, through a specific set of APIs for component instantiation, binding, life-cycle management, etc. Usually, the set of allowed modifications is constrained by the types (including component and interface types), and the architectural style of the involved components (see ArchJava [30], Acme[31], SOFA 2.0 [32] for instance). More recently, requirements related to an associated meta-model have been explored, e.g. [33].

Besides using directly an API, the modifications can also be expressed through scripting languages, associated to the ADL like in Darwin, ArchJava, or built upon the API as FScript [12] for Fractal components, Lua for OpenCOM [34], or GCMscript (see Section 4.1). Allowed modifications can also be the result of some component associated constraints (Architecture Constraint Language) as in [35], or reconfiguration rules (Event Condition Action rules) like in Automate [36], Plastik [37], Safran [38], Dynaco [39], or Rainbow [40], most of them relying upon the MAPE model for autonomic computing [26].

In this context, it is valuable to be able to check beforehand and to ensure that the configuration resulting from a programmed or rule-driven modification will comply to the architecture constraints and correspond to the desired global-state. To reach this goal, some works check in advance that some independently developed reconfiguration rules are compatible [41], or even enforce the correct synchronisation of different autonomic control loops [42]. Some approaches check that, given a behavioural specification for each involved component, the resulting composition features desired properties (e.g. no deadlock can occur even after reconfiguration operations). The Vercors verification platform [43] is able to check this kind of properties for GCM/ProActive components. Some works ensure that in case some reconfiguration process starts but can not fully lead the architecture to the required state, e.g., if the reached state violates some invariants of the component model or of the application, then, the process is aborted and the architecture is kept unchanged. For example [44] and [45] rely on transactional managers to maintain some predefined invariants about the component model, the architectural style, and the application. Some approaches regularly check whether the current status is compliant to some predefined invariants and, if it is not the case, then some automatic reconfiguration is triggered. For example, Acme/Armani [46] expresses architecture invariants with first order logic.

Reconfiguration and Hierarchy Hierarchical component structure plays a particular role relatively to reconfiguration (see [32] for a detailed discussion on this aspect). Not all component models makes sure that the hierarchy is also available at runtime, once leaf components have been instantiated. In particular, models that do not allow introspection and dynamic reconfiguration of the component architecture do not consider useful to represent composite component instances at runtime. As an example, this is what a strict interpretation of the SCA specification [8] would imply, because SCA scope is only design time and excludes component deployment and instantiation, leaving them as a platform-specific concern. On the contrary, in FraSCAti [11], because it is an SCA implementation based upon Fractal, it is useful to instantiate FraSCAti composite components; this leads to the only SCA platform offering run-time management of SCA assemblies.

2.1.3. Open component models Component models can be considered highly configurable if it is possible to choose the way the application is controlled through policies and/or meta-level artefacts constituting component container or management level. Some components even allow the programmer to define his own control interfaces; for example we will see in this paper how

DOI: 10.1002/spe

controllers can be added to GCM components (see Section 4.2). Those component models that allow the customization of the set of available managers are called *open component models*.

Whereas EJB, .NET, CCA [9] rely on quite strict predefined containers/controller sets, open container based models like CCM [47], or reflective component models like OpenCOM [34] and Fractal [21] are highly configurable. Ultimately, by plugging the adequate additional controllers to such generic component models, the component architecture can be made self-reconfigurable. For example, Jade [48] is a self-repairable component framework built upon Fractal; more general-purpose self-adaptation is provided by, for example, Automate [36], Safran [38], Dynaco [39]; the last two frameworks also built upon Fractal. In Section 5.2 we will illustrate how GCM has been adequately extended to provide self-management for component applications.

2.1.4. Adaptable control level Moreover, the meta-level/management layer itself could be dynamically reconfigurable. The advantage of such a paramount flexibility degree is to allow the adaptation of the control strategies, without having to redeploy the whole application; this is particularly useful in case the context in which the components are running evolves so drastically that even the control layer needs to be adapted.

To enable the dynamic reconfiguration of the control layer itself, this layer should be programmed in a flexible manner. Component orientation allows such level of flexibility, yielding a *componentized membrane* concept as presented in [20]. Besides Fractal/AOKell, a Fractal implementation [20], the SOFA component model whose control part is built using microcomponents [10] and Dynaco that uses full-fledged components [39], in GCM and in its implementation presented in this paper such flexibility is achieved by the fact that the control level is also made of components (see 3.3).

An alternative to implement a non-functional concern (i.e. a code orthogonal to the functional code) is to rely upon dynamic AOP techniques (as in Safran [38]). However, in this case where the concern is an aspect and not a component, it suffers the known drawback of AOP: the order aspects are executed is arbitrary (i.e. dictated by their attachement order), and they are not able to interact. On the contrary, a concern implemented as a component allows it to be manipulated as a first class entity within the component-based software architecture forming the control layer: collaboration with other concerns to fulfill its dedicated task becomes thus possible.

2.1.5. Techniques for component adaptation As discussed above, either the functional or the control part of the components can be adapted. However, one should also distinguish the different techniques used to perform adaptation. Depending on the chosen technique, adaptation can be more or less expressive and more or less intrusive on the executed code. Indeed, three levels of intrusion can be identified: first, adaptation can modify the component architecture by adding or removing components, or changing the component connections; second, adaptation can influence the behaviour of primitive components by adjusting parameters of the components; finally, adaptation can change the executed code itself. This third level is generally achieved by techniques such as AOP, like in the Safran framework. Note that all three levels of adaptations are meaningful both for functional and for non-functional adaptations.

In our approach, we rely purely on component and object techniques; this has been sufficient to achieve adaptation in the two first levels in a quite natural and efficient way. Regarding the third level, our implementation also relies on a meta-object protocol allowing us to reify, intercept, and manipulate inter-component communications. This gives us and intermediate expressive power between pure component manipulation and full control of the application code granted by AOP techniques. For example, our interception mechanism allows us to perform monitoring of business method invocations by generating events that are sent to the autonomic control layer.

2.2. Distributed component models

Among the numerous component models, we focus below on those specifically targetting distributed computing. Also, we exclude models such as Java Beans speciallly geared towards graphical user oriented applications, EJB used within the application server tiers of application servers, OSGi a

DOI: 10.1002/spe

centralized (mono JVM) service oriented component model which focus on dynamic loading of service implementation and dynamic service resolution. In this paper we focus on models where at least some of the components can be deployed on several nodes. This is exemplary of SOA applications where component/service instances are de facto quite coarse grained and decentralized because they can be made available by different parties spread at the Internet scale; and of distributed applications using several nodes to distribute communication and computation load. In this last case also, the granularity of components is not very fine, because the overhead of distribution would be too high if every single small component could be accessed in a distributed manner.

Distributed component models should also be convenient to represent parallel applications. We call *parallel application* an application that can be modelled as a set of quite identical components that interact frequently to share data needed for the computation. In this case, besides one-to-one interactions, interactions between multiple components (i.e., many-to-many interactions) are also required. This is why models targetting parallel computing introduce collective interfaces including MxN interfaces as in CCA [9], GCM [1].

Components as an abstraction for Distribution We believe that the component abstraction is a good programming abstraction, both at deployment-time and at runtime; this is why we use the component as the unit of distribution: a single primitive component should be placed on a single machine. Of course, a composite component consisting of several others can be distributed over several nodes; and similarly if the control of a component is made of several components, then those components can be spread on many nodes. Components serve as strong structuring and deployment entities, both at the functional level, and at the control level.

The first interest of having such an abstraction is the benefit from hierarchy as a way to handle component distribution. A major goal of distributed component models is to target large-scale distributed computing, where the number of interacting distributed components is high. Because these components are distributed, it is intrisically complex to manipulate all or specific subsets of them at once. For example, consider a subset of components contributing to a parallel computing application and deployed on the nodes of a given computing cluster. Assume this cluster is going to be shutdown for maintenance purposes, the management layer needs to trigger the migration of the whole component subset. Having an abstraction that gathers all the distributed components currently deployed on the cluster would ease the execution of the global migration. Another example comes from SOA, where all the components that contribute to implement a coarse-grained component oriented service, be they distributed or not, have to be considered as participating to the same entity. For those reasons, hierarchy eases the management of distributed components.

If hierarchy is to be explicit at runtime and if components that belong to the same composite component are distributed, then the composite component itself should be instantiated at a chosen location. Sometimes, especially when components feature an autonomic behaviour, the control part of the component is so complex that it becomes useful to spread the control part of one component on several nodes. In this case, all software artefacts in charge of the component management can themselves be also spread on many nodes.

To our knowledge, the approach we advocate in this article is the only one to feature this expressive power: in GCM/ProActive, control and management of a component can itself be realised by a distributed system. Another particularity of our approach is that we adopt a strict and well-defined execution model adapted to the distributed nature of our components.

2.3. The special case of service oriented component models

Service oriented component models and platforms make it totally explicit that the aim of components is to offer services, which are central in SOA. Along the SOA paradigm, services can exist independently of any client; they are present in the environment, and the environment must allow clients to dynamically discover, select, and bind to these services for further invoking them. Of course, clients themselves can be applications built around a component oriented model, that can, for example, express service requirements through what is named an utility interface [10], but this is not mandatory.

In service oriented computing, services are commonly stateless, allowing many service invocations to be executed concurrently by the component implementing the service. However, services can also be stateful, meaning that care has to be taken within the component-based service implementation whenever more than one service invocation is to be served.

Some component models for SOA such as the SCA specification and its implementing platforms like Tuscany, Newton, or FraSCAti, allow components to be implemented using any programming language, as Java, Cobol, etc., including BPEL, the standard web services orchestration language. In BPEL, it is explicit that the role of any component is to *orchestrate* some required services, resulting in the provision of new exposed services sometimes also registered for further discovery and used by other components/services. Allowing heterogeneity in the way components are implemented has also some consequences on the communication protocols allowing such heterogeneous components to interact. This explains why SCA bindings can be selectively bound to a particular communication protocol such as SOAP, Java RMI, Sun JMS, etc. As shown by FraSCAti and also by SOFA 2.0 [25], the challenge of these SCA platforms is to support such a high variability level in the technologies available, in particular concerning communication connectors that requires a very modular and flexible infrastructure.

2.4. Positionning: Programming Distributed Components

Our original contribution relies on a strong interconnection between a component model and a programming language. The main strength of GCM/ProActive is the coherence between the composition, the execution, and the distribution structure for the application: components are well separated entities that can be placed on different machines and execute independently. The closest works from us are the other formally specified programming models that provide support for components: Creol [49] and ABS [50], which is inspired from Creol. In Creol, components are extremely simple, non-hierarchical, and with little support for binding and reconfiguration. In [50], the authors take a process calculi perspective for designing components, which is well formalised but much further from the software component models than the approach we advocate here. In any case, the programming languages ABS and Creol are very close to our programming paradigm and are also formally specified (see [51] for a formal specification of GCM/ProActive). These works also strengthen the idea that the combination of active objects and components enable the effective design and programming of large-scale distributed and autonomic applications. Most component models focus on the structural aspects, but without information on the execution and synchronisation model. It is then extremely difficult to predict the behaviour of a component system. Indeed, predicting the result obtained when running an application made of components relies on a precise knowledge of the way these components interact and synchronise.

From a more practical point of view, every implementation of a component model relies on one or several programming languages and interaction paradigms. The practical choice made in real implementations (Julia for Fractal in Java [21], MOCCA for a CCA framework in Java [19]) imposes at least a language for defining the interactions between components. However, there was no previous study on the interplay between the execution and the composition models. We claim that having a precise execution model for components is crucial in order to program them safely and efficiently. Of course, several component models like CCA, SCA, or CCM also allow the connection of components written in any programming language, and with a lot of variations on the interaction protocols. In a practical component infrastructure, it is crucial to take into account the connection with off-the-shelf components. That is why in GCM/ProActive and in this paper we put a particular focus on the interconnection of our components with other components specified as services. Thought the interconnection of our components with the SOA world may partially break the coherence we created in our composition and execution model, we think this integration is necessary for programming real component systems. We will see in Section 5.3 that, through SCA, we manage to let our programming and execution model interact with service oriented components in a convenient and effective way, while the pure GCM/ProActive part of the application comes with a strong and well-specified execution model.

In conclusion, existing component models do not sufficiently consider the interaction between the application architecture and its runtime behaviour. Our component framework includes a programming and runtime model that is suitable for developing and deploying complex applications especially in the context of autonomic, service-oriented, and large-scale distributed computing. The rest of this paper presents our component model, runtime model, and reconfiguration and autonomic framework; these contributions lead to a programming and execution model where autonomous composition and hierarchy is present in every concern and at every stage of the component lifetime.

3. PROGRAMMING DISTRIBUTED COMPONENTS: GCM AND ACTIVE OBJECTS

3.1. Active Objects

Programming distributed applications is a difficult task. The distributed application developer has to face both concurrency issues and location-related issues. The active object paradigm [52, 7] provides a solution for easing the programming of distributed applications by abstracting away the notions of concurrency and of object location. Active objects are similar to actors [53], but better integrated with the notion of objects. Active objects are isolated entities that are manipulated by a single thread; they communicate between themselves by asynchronous method invocations. Active objects act as the unit of distribution and of concurrency, naturally abstracting away communication between remote entities and local availability of data. An object is said to be active if it is equipped with a thread and a queue for storing invocations; it can be deployed on a remote machine. As a consequence, every method call to such an object will be a remote method invocation; we call such an invocation a *request*. An active object is thus an object that treats the requests it receives; it is an object together with a thread. To ensure absence of sharing between activities, each object belongs to a single activity led by a single active object.

To decouple the caller object from the invoked object, contrarily to a classical remote invocation, the invoker is not blocked waiting for the result, instead a *future* object is created and represents the result of the remote invocation. A *future* [54] is a placeholder for an object that is not yet available; this construct is very convenient for easily expressing concurrent or distributed programs. Futures can be stored and passed as arguments or results of (remote) method invocations. In ProActive, we chose to have transparent futures so that the programmer does not have to explicitly manipulate futures: if a future object is accessed while the object is not yet available, the program is blocked. The advantage of this approach is that while synchronisation between entities is simple, the programmer does not have to worry about which objects can be a future, and the result of a remote invocation is only waited at the moment when it is really needed.

Next section describes the structure and execution model of GCM components. It presents the active object execution model in the context of the GCM component model.

3.2. Asynchronous Components

GCM is a component model adapted from Fractal. The structure of a GCM component assembly and the terminology used by GCM are shown in Figure 1. Services are offered through *server interfaces*, and required services should be bound to *client interfaces*. The first architectural particularity of GCM components, compared to Fractal, is the existence of one-to-many (multicast), many-to-one (gathercast), and many-to-many (MxN) interfaces. One-to-many interfaces transform a single invocation into many invocations, potentially distributing the invocation parameters. Many-to-one interfaces wait for several invocations and transform them into a single one, potentially aggregating the parameters received. MxN interfaces feature both aspects. More details on collective interfaces can be found in [55]. The second architectural particularity of GCM is the way non-functional concerns can be structured inside what is called a *membrane*; this will be detailed in Section 3.3.

GCM/ProActive is the implementation of the GCM component model based on active objects. In GCM/ProActive each component is implemented by an active object so that the component architecture reflects the distribution and the concurrency aspects. In the following, each time we mention component we are referring to a GCM/ProActive component that contains an active object.

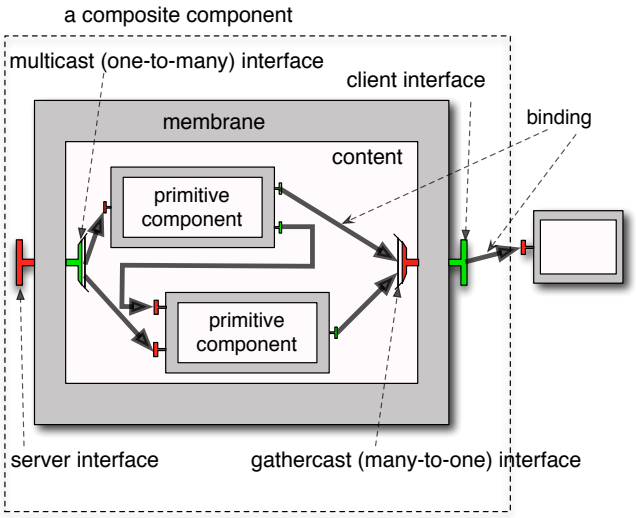


Figure 1. A GCM component assembly

One of the main advantages of using active objects to implement components is their adaptation to distribution. Indeed, active objects provide a natural way to provide loosely coupled components. By loosely coupled components, we mean components responsible for their own state and execution, and only communicating via asynchronous communications, i.e. asynchronous requests with futures. Asynchronous communications increase and automate parallelism; and absence of sharing eases the design of concurrent systems. Additionally, loose coupling reduces the impact of latency and enforces the fact that each component is responsible for its own execution. Finally, independent components also ease the autonomic management of component systems, enabling systems to be more dynamic, more scalable, and easily adaptable to different execution contexts. That is why we think that a distributed component system should rely on loosely coupled components communicating via asynchronous communications, and not sharing any memory; consequently, we think active objects are particularly adapted to implement a distributed component model.

One of the interests of ProActive is that no specific syntax or instruction is needed to program active objects compared to usual Java code. Consequently, the content of primitive GCM/ProActive components is similar to Julia components [21]; it consists of classical Java code. The interested reader could find in [56] examples of ProActive programs, and of GCM/ProActive components. In ProActive, as no code is needed for distribution, the non-expert programmer can reuse its application written in pure Java (or in Julia). However, the expert programmer might also simply want to reorder the instructions involving communications and synchronisations, e.g. separating the remote invocation from the access to the result, in order to increase parallelism.

Communication Active objects provide a convenient programming model for distributed components; this implies that components communicate as follows. Components are loosely coupled and communicate by asynchronous message sending: when a component performs an invocation on its client interface, this invocation is transmitted to a destination component and the message, called *request*, is enqueued at the receiver side in a queue. The message will be treated later by the destination component. Request parameters are copied when passed between components, so that each component remains responsible for its own state.

Asynchronous treatment of requests is a crucial feature of our model as it allows each component to execute independently. The use of futures allows the invoker to keep a future reference to the result and to only block if it needs the computed value. Futures can be passed by reference between components, as invocation parameters or results.

DOI: 10.1002/spe

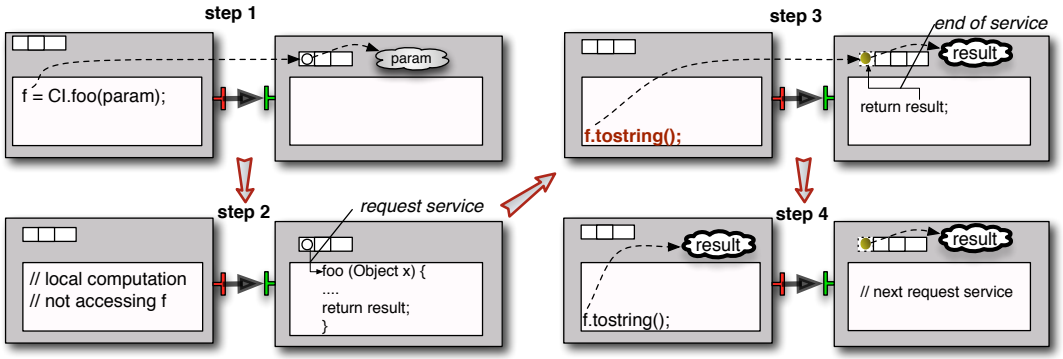


Figure 2. A simple communication scenario

Components do not share memory. The only shared references are references to futures that have a single-assignment semantics; their value will be finally transmitted by a copy semantics. Consequently, there is no concurrent memory access between any two components. Overall, each component has its own thread, executing independently from the other, in a much service-oriented manner. Of course, several components can also be bound to the same destination component and invoke its services in a concurrent manner.

Execution scenarios Let us now illustrate the execution of GCM/ProActive applications by a couple of typical scenarios.

Figure 2 shows a simple communication scenario between two GCM/ProActive components; it consists of four steps. The configuration simply consists of two primitive components bound together. `CI` is the name of the client interface of the component on the left; it is bound to the single server interface of the component on the right. Each component contains a request queue (set of boxes in the top part). At the beginning (step 1), the component on the left performs an invocation on the other one; this creates a new future and enqueues a new request in the destination request queue (this is shown by a circle at the first position in the request queue of the component on the right). Then, the component on the left can perform local computations or other remote invocations; after a while the component on the right will serve the request and execute the associated business code (step 2). Possibly, the component on the left may access the future `f`, blocking the execution until the result is computed and returned; in the meantime, the component on the right executed the request and computed a result for it (step 3); the request is now finished and a result is associated to it. The left part of the request queue is shown with dotted borders to indicate that the requests in this part of the queue are finished and a result value is associated to them. Finally, in step 4, the result is returned to the component on the left that can continue its execution.

This example shows the loosely coupled nature of the component model and the asynchronous execution that results. We call our component model asynchronous because communication does not trigger computation on the receiver side immediately, it just enqueues a request. However, such a mechanism can be implemented with synchronous or asynchronous communications. In GCM/ProActive a rendezvous ensures the causal ordering of messages [57].

Figure 3 shows a slightly more complex scenario where a component delegates the service of a request to another component, illustrating the importance of futures and of the fact that they can be passed as reference between components. The configuration consists of three primitive components, **A**, **B**, and **C**, bound together. In step 1, when the component **A** makes an invocation to component **B**, this one delegates the invocation to **C**. The new request is called `gee` with a modified invocation parameter and the result of this delegated invocation is directly returned as a result. Step 2 shows that, as future references can be passed between components, the request service `foo` in **B** can terminate and **B** can now serve other requests from its queue while **C** is made responsible of replying to component **A**. Finally, in step 3, the future reference is returned to **A** that now directly waits for

DOI: 10.1002/spe

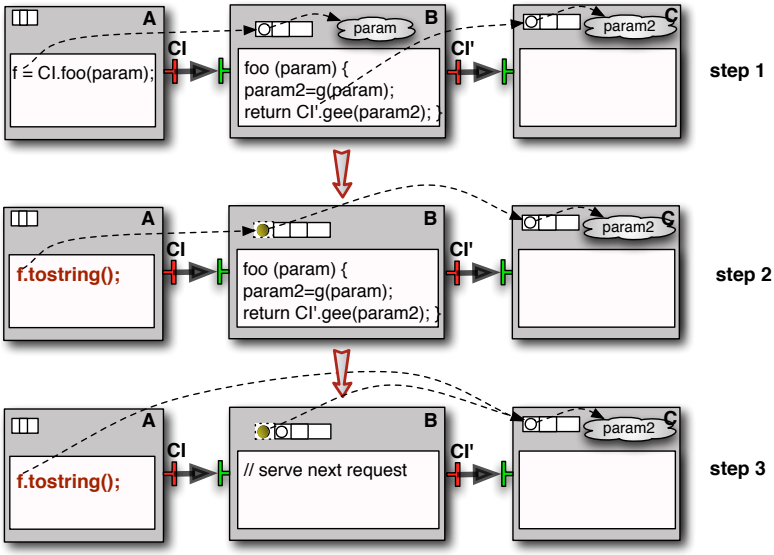


Figure 3. A delegation scenario

the result being computed by component **C**. Eventually, **C** will serve the request and compute a result value for it. The computed value will be transmitted to each component that holds a reference to the corresponding future; in this case, to components **B** and **A**.

Hierarchy GCM is a hierarchical component model. Primitive components are the leaves of the composition tree; they contain the application logic. By contrast, composite components are made of other components, either primitive or composite ones. Composite components have a predefined behaviour because they are only used as composition tools: they serve all the requests they receive and transmit them to their internal components or to external components depending on the invocation and the bindings. Composite components are also implemented by active objects and equipped with a request queue, but the service of a request is predefined: a composite component keeps delegating invocations in a much similar way to the component **B** of Figure 3. In other words, service methods are generated automatically. Requests follow bindings, going from an emitter primitive component to a receiver primitive component that is bound to it; this binding can be an indirect binding, passing through the bounding box of several composite components. Implementing composite components with active objects allows us to have a programming model (and a behaviour of components) common to all the hierarchical levels. This is particularly useful when implementing the control part of the components: composite components can be monitored and controlled in the same way as primitive ones. For example, a composite component can be stopped and still accept incoming requests in its queue (in order to serve them after when the component is started), thus preventing the invoker from being blocked.

Figure 4 illustrates the flow of invocations in a composite component **A** that surrounds a primitive component **B**. **B** has one server interface and one client interface, both bound to **A**. A request is received by **A** on its server interface `SI` (step 1). This request is handled by **A** and sent unchanged to the sub-component bound to the interface `SI`, component **B** (step 2). Then, suppose that **B** relies on an external invocation to serve the request. This invocation is sent to the surrounding composite **A** in step 3. In the meantime, component **A** has finished the service of the first request and returned a future pointing to the request in **B**, similarly to the delegation scenario above (in Figure 3, the component **B** returned a future as result). Then, symmetrically to step 2, the service of the request by the composite only consists in sending the request to the external world through its external client interface `CI`. Note that the composite is able to serve the second request because the first one

DOI: 10.1002/spe

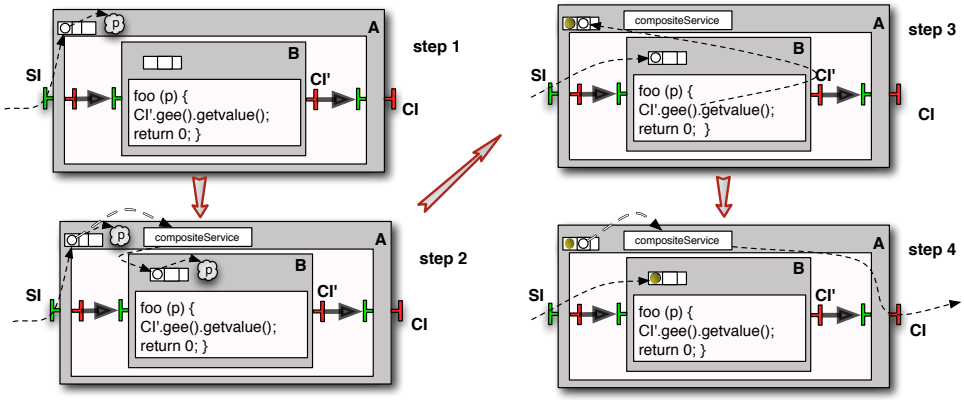


Figure 4. Delegation in a composite component

is finished. Note that, contrarily to **A**, component **B** is waiting for the result of the invocation on `gee()` and will only be available when a real result for the request on `gee` is available.

Granularity of the component model A general issue when designing a component model is the foreseen granularity of the components. In the case of a hierarchical component model, this question can be refined into "what is a good size for a primitive component?"

Fractal does not impose any granularity for the components, but the existence of composite bindings and some of the features of the model suggest a rather fine grained implementation: a primitive component should contain a small number of objects. In GCM/ProActive in order to allow primitive components to be the unit of distribution, GCM/ProActive components should have a coarser granularity than Fractal ones. Overall, the GCM has been conceived with a granularity that is somehow in the middle between small grain Fractal components and very coarse grain component models, like CCM where a component is of a size comparable to an application. Somehow, GCM has been conceived thinking of components of the size of an MPI process (or an active object); however, it can also be used in a finer or coarser grain way.

This difference of granularity between Fractal and GCM partially explains why some of the features that could be implemented by a small Fractal component and are highly used in a Grid setting have been defined as first class citizens in the GCM. For example, multicast interfaces could be expressed in Fractal by binding components that perform the broadcast, but such components would be too small to be used as the unit of distribution.

This granularity argument also explains why we can afford to attach the structure of an active object to each component; if a component only contained a few objects, it would not be reasonable to allocate a thread, a request queue, and a future pool to it. Consequently, GCM/ProActive is not designed to create a large number of components on the same machine. If a distributed application is to be composed of a very large set of small components, then it is more efficient to design it as a GCM/ProActive component system of medium-size components where each of the GCM primitive component consists of a Fractal assembly of small components.

Advantages and drawbacks of the execution model By nature, our model enforces that each component owns a single thread, which imposes a component structure that fits the distribution and concurrency aspects. Our execution model enforces a strong separation between the execution inside each component. Consequently the design of the application is made easier. Thanks to the mono-threaded nature of each component, no race condition is possible, and there is no need for locks or synchronization code. Components only interact by request-replies on well-defined interfaces and thus each component can be programmed really independently from the others.

Futures help in the transparent management of values returned by a service invocation through a component interface. They dramatically ease the programming task by allowing the content of a

component to appear as a non threaded code relying only upon (remote) methods as in an object-oriented non concurrent program. Futures by themselves do not improve performances. Getting performances (high parallelisation) comes from the asynchronous (and distributed) processing. Asynchronous processing could be obtained through a hand-written concurrent and distributed approach (e.g. using explicit thread programming, making sure distributed calls are asynchronous and corresponding replies come back to callers through explicit callbacks as in the actor model). With our approach, the code is looking non concurrent and so it is more easy to understand, but it is still parallel thanks to asychronous calls with futures offered by the active object technology.

The main drawback of this approach is that the synchronization that occurs when accessing a future can create deadlocks in case of circular dependencies. For example, if two components are mutually dependent, and the first one is waiting for a result from the second, while the second needs a reply from the first one before continuing its execution, then a deadlock occurs. The risk of such deadlocks is however limited by the fact that futures can be transmitted as results or invocation parameters. Indeed, if in a circular dependence one of the requests directly returns a future, then the deadlock can be avoided, as illustrated in the composite component case (see Figure 4 and its explanation above). A deeper analysis of the use of transparent futures, their advantages and drawbacks, and the design of a static deadlock detection framework for components with transparent futures can be found in [58].

We are investigating the possibility to allow some controlled multithreading inside active objects [59]. Applied to components, this approach should remove some of the deadlocks (in the example above, the first component could create a new thread in order to answer to the second one), but this aspect is outside the scope of this article. Additionally, these multithreaded components partially overcome the limitation of one single thread per component, allowing us to better scale on multicore architecture: one component is able to run several threads on the same machine.

Finally, one of the main advantages of our model is that it is easier to design autonomic strategies for the management of such loosely coupled components, as we will show in Section 4.

3.3. Componentized Membrane

Fractal considers the control part of a component as a set of *object controllers* contained in the membrane of a Fractal component. These controllers are defined in a static way when the component is instantiated and they are accesible via *control* interfaces.

NF Components GCM extends the notion of *controllers* to allow the introduction of Non-Functional (NF) components [60], in line with the componentized membranes introduced in [20]. NF components reside in the membrane (c.f. Figure 1) of GCM components and have the same goal as object controllers: they provide management features to GCM components and take charge of NF tasks. We call such a construct a *componentized membrane*, it enables the component-based design of component control. Because NF tasks can be complex to program and even may need adaptation to face to changing runtime conditions, adopting a component-based architecture is a very flexible and powerful solution. It also provides better reusability. Other than that, NF components share the same features as regular functional (F) components. They can be assembled, composed and introspected, and even can have NF components in their membranes as well. The possibility of having a componentized membrane brings some advantages to GCM. Notably, (1) the separation of NF concerns from the functional part by decoupling the F and NF implementation, and (2) the possibility of easily designing and implementing complex and adaptable NF behaviours.

The separation of concerns between F and NF parts allows us to design the two behaviours in separate ways and delay the integration of the two aspects until deployment time. Even then, after the component has been instantiated, it is possible through introspection to analyse and dynamically modify only the F components, or only the NF components without affecting the rest of the architecture. This decoupling permits the management concerns to be designed by an expert that does not need to know about the functional behaviour, and, conversely, the programmer of the functional part needs not to focus on the non-functional concerns. Concerns like composition, bindings, monitoring, reconfiguration, security, load balancing, and in general self-management

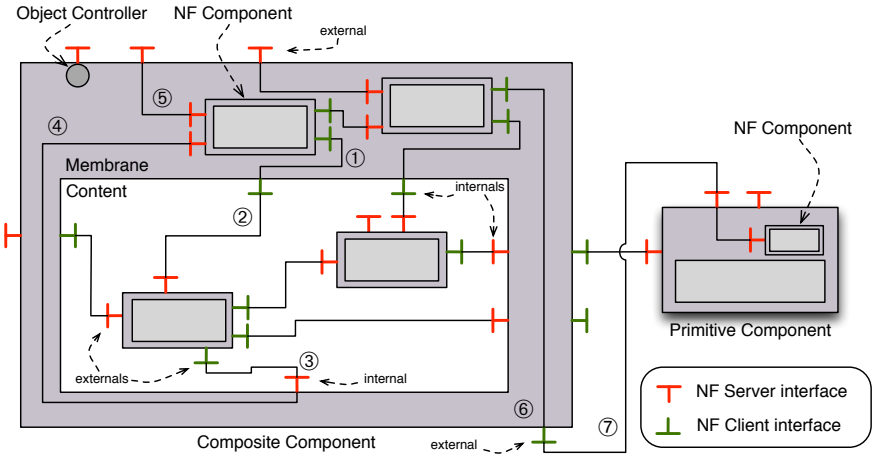


Figure 5. Composite GCM component with components in the membrane

behaviours can be designed without having to know the details of the functional part, focusing only on the component management. However, at execution time, both F and NF parts need to interact. In particular, a NF concern related to self-management may need to know the composition or to observe the behaviour of the F part, and it may need to modify some attribute or to carry on an action that affects the F part. For this effect, certain NF interfaces like the *attribute controller*, *binding controller*, or *content controller* allow the observation or the modification of the characteristics that affect the behaviour of the F part (but only the business code exposed by the component, i.e. only the one accessible through the component API). It is also possible to add customized implementations of these NF interfaces or to add additional NF interfaces.

Some NF concerns like self-management behaviours may require a complex implementation, and it becomes reasonable to separate the application control in different components. By using a componentized membrane we allow the implementation of complex tasks using components that, similarly to the functional part, can be assembled, composed, and replaced.

Of course, this does not forbid to have regular (non-active) objects implementing NF concerns. In general, the most basic tasks like the control of the life cycle of components, the control of bindings, and the control of the compositions, do not need the complexity and overhead of a component oriented approach. The componentized membrane is flexible enough to allow different levels of complexity by implementing some NF concerns using non-active objects as controllers and leaving others as components.

GCM/ProActive provides an additional advantage. By implementing NF concerns as GCM components, we may have parallelism and asynchronous communications inside the membrane itself improving the efficiency of the NF tasks. It is even possible to transparently distribute the NF components themselves.

Figure 5 shows the GCM notation for the componentized membrane. The figure shows two components in the membrane of a composite component, and one component in the membrane of a primitive component.

NF Interfaces and NF bindings If we want to develop NF tasks that require the coordination of different membranes, this is not possible by the traditional Fractal model, where there is no explicit binding between NF interfaces. GCM provides client NF interfaces, which enable inter-membrane bindings and communications. We first review the different kinds of NF interfaces that exist in GCM, and then detail the set of bindings that can interconnect them, illustrating the necessity of such NF bindings to realize non-trivial management procedures.

In many cases, management activities require a meaningful collaboration of several tasks that can be spread transversally to other components. For example, if we want to get a measure of the energy consumption of an application, it is necessary to aggregate the energy consumption of all

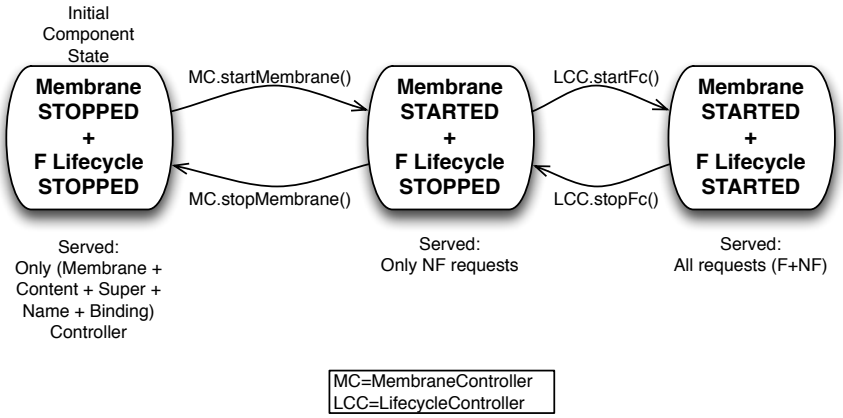


Figure 6. Lifecycle of a GCM component and request service policy

the individual components. In a different case, if we want to provide a security certificate for an application, this certificate must be updated through all its components.

GCM defines control interfaces, referred to as *NF interfaces*. Where Fractal components provides only *NF server interfaces* (called *controller* interfaces), GCM also includes *NF client interfaces*. NF client interfaces can be bound to NF server interfaces of other components, in order to communicate with other membranes and allow collaborative NF tasks. Collaborative NF tasks can also be created between the membrane of a composite component and the membranes of internal subcomponents following the hierarchy of GCM components. For this case, GCM defines additional *internal NF interfaces*. Internal NF interfaces allow components residing in a membrane to communicate with the membrane of components residing in the functional content, and conversely, it allows the membrane of internal components to communicate with components in the membrane of the parent.

Figure 5 shows internal and external NF interfaces, and the possible interconnections between them. Figure 5 also illustrates the possible bindings using NF client interfaces. A client interface of a NF Component can connect to an *internal NF client interface* that pertains to the membrane (1). This *internal NF client interface* acts as a passage from the membrane to the content. An *internal NF client interface* can bind to the *external NF interface* of a sub-component (2). This allows the propagation of management tasks originating from the membrane to the membranes of sub-components. A simple example is the sending of a stop signal from the membrane of a composite to the internal components.

The communication can also flow from the content to the membrane. An *external NF client interface* of a subcomponent can connect to an *internal NF server interface* of its parent component (3). This *internal NF server interface* is a communication channel from the content to the membrane. The *internal NF server interface* can also connect to the server interface of a NF Component (4). This allows the membranes of sub-components to access to management tasks on the membrane of its parent component. For example, a component that represents a storage resource can use a NF client interface to communicate to its parent that the storage capacity is near full, so that the membrane of the parent can take some action.

The other elements shown in Figure 5 show the bindings between the *external NF server interface* of a component and the server interface of a NF Component in the membrane (5); the bindings between the client interface of a NF Component to an *external NF client interface* (6) to allow communication with external components. Finally, an *external NF client interface* can be bound to an *external NF server interface* of another component (7), effectively interconnecting the membrane of two components. For example, a security manager residing in the membrane can propagate a security certificate to all the components with which it maintains a functional binding in order to authenticate itself.

3.4. The Component Lifecycle, and the Request Service Policy

The component lifecycle and the associated scheduling of requests deserves some additional explanations. As explained in Section 3.1, a component has a single request queue. This queue receives both F and NF requests and serves them in a sequential way. However, there are situations in which it is useful that the component serves only NF requests while leaving the F requests in the queue; e.g., this is the case for the dynamic adaptations primitives that will be presented in Section 4.

The lifecycle of a component, and the requests served in each state are shown in Figure 6. Both the non-functional part and the functional part can be either in the *started* or in the *stopped* state. A component can only be in three different states concerning its lifecycle: either it is fully started (both membrane and functional part started), in which case it serves all the requests in its queue one after the other. On the contrary, a component can be fully stopped, i.e. both its membrane and its functional content are stopped; in that case, the only requests served are those on the following object controllers: *MembraneController*, *ContentController*, *NameController*, *SuperController*, and *BindingController*. There is an intermediate state where only the membrane is started; in this state, all the non-functional requests are served, including the ones targeting components in the membrane or subcomponents. The lifecycle state of the component can be changed by invoking the appropriate method (labeling the arrows in Figure 6) on the *LifeCycleController* or the *MembraneController*.

Additionally to this service policy, each non-functional request is attached a priority level; three priority levels exist. Request of the highest priority level will be served before any other request; this is useful for enforcing the rapid enactment of a critical reconfiguration step. Requests of the intermediate level will be served before any functional request that is before it in the request queue (maintaining the order relatively to the other non-functional requests); this is useful for performing several non-functional operations consecutively. Finally the lowest priority level is the default one, it treats the requests in a FIFO order, even relatively to functional ones maintaining the potential causality between the different requests.

4. DYNAMICITY IN GCM

This section focuses on the “dynamic” part of GCM/ProActive. This is, how an initially deployed architecture may be modified at runtime, without the need for stopping all the system. Initially we provide a set of primitives and a reconfiguration language that allows us to describe and execute reconfigurations. Later, we explain how this can be executed autonomically by inserting components in the membrane that take care of the autonomic behaviour and that these components can also be modified to change dynamically the autonomic behaviour. In the last section we unify both features, and mention how these two features allows GCM/ProActive to be used to build self-adaptive applications.

4.1. Reconfiguration

Like Fractal, each GCM component provides control interfaces giving the possibility to modify, at runtime, the application structure. Those interfaces can be invoked to add or remove components or bindings at runtime. Generally, for consistency reasons, it is required that the components directly impacted by a reconfiguration action are stopped before being reconfigured. Unfortunately, directly invoking those reconfiguration procedures can be cumbersome: accessing components in the hierarchy requires multiple introspection calls, and multiple exceptions can be raised; consequently the code actually doing the reconfiguration is just a small proportion of what has to be written.

To ease the programming of reconfiguration procedures we designed a framework dedicated to the reconfiguration of distributed components, and in particular to the reconfiguration of GCM components. When studying the adequacy of languages dedicated to the reconfiguration of components to GCM, we realised that FScript [12] was close to be adequate but was too much centralised. Indeed, FScript provides the primitives that are necessary to reconfigure a component system made of Fractal or GCM components through reconfiguration scripts; it ensures a much

DOI: 10.1002/spe

higher-level of abstraction than direct invocation to the Fractal or GCM API, but the reconfiguration scripts were designed to be interpreted in a centralised manner.

In a distributed component model, it seems more reasonable to interpret reconfiguration scripts in a distributed manner: several composite components can be responsible for reconfiguring their sub-components independently. This distributed approach allows parallelism, and thus allows reconfiguration procedures to be run on large-scale infrastructures. It is also better adapted to program autonomic adaptation procedures, as each component can embed the adaptation scripts necessary for its adaptation and trigger them autonomically, when needed. Allowing each component to embed its own script interpreter, and to interpret its own local scripts also contributes to the loosely-coupled nature of our component model.

For this, our approach was quite simple but really effective, it is based on two extensions of the FScript framework. The resulting reconfiguration scripting language is called GCMScript, and is integrated with GCM/ProActive.

A controller for reconfiguration We added a non-functional port, localised in several (possibly all) components. This port is able to interpret reconfiguration orders. We called the non-functional object able to interpret reconfiguration scripts a `ReconfigurationController` and embedded it inside the membrane of the desired components.

This controller embeds an instance of the FScript interpreter and provides a method `load` for loading reconfiguration (sub)scripts, and a method `execute` for triggering a reconfiguration action.

In general, to ensure consistency and to avoid unpredictable interplay between execution and reconfiguration of a component system, it is necessary to stop the functional behaviour of a component while reconfiguring it. Indeed, the outcome of a request sending through an interface, while this interface is being unbound, is unpredictable. In previous works, we designed a protocol able to stop an asynchronous composite component together with all its subcomponents in a safe way [61]. The objective of such a protocol is to ease the design of safe adaptation procedures: when a subsystem is entirely stopped, it can go through a reconfiguration phase before being restarted. As explained in Section 3.4, stopping a component does not stop the membrane and thus a stopped component can still interpret reconfiguration scripts and be reconfigured.

A primitive for distributed reconfiguration We extended FScript with primitives for triggering the remote execution of reconfiguration scripts. The primitive `remote_execute` triggers the execution of a reconfiguration action on a remote component. The target component is specified as a FPath expression [12]; it is the component that will evaluate the reconfiguration action. FPath is a domain specific language to specify a path inside a component assembly; it is embedded in the FScript interpreter and eases the exploration and reconfiguration of components inside a possibly complex assembly. Upon remote script invocation, if no remote interpreter is available then one is automatically created by calling the `setNewEngine` method on the remote reconfiguration controller. Then, the target component becomes in charge of interpreting the reconfiguration. Then the calling interpreter can continue the execution of its local script. Each reconfiguration script then runs independently. The script invocation creates an asynchronous invocation (without result).

Figure 7 illustrates the principles of the distributed interpretation of reconfiguration scripts written in GCMscript. Initially, the configuration consists of two components: **A** and **B** which is inside **A**. Both components are equipped with a *reconfiguration controller* non-functional interface and embed a GCMscript interpreter. When the action `main` is invoked, the associated script is interpreted locally at component **A**; this action consists of two instructions, the first one (1) corresponds to line 2 of the script; it puts the component **C** previously instantiated inside the component **A**. For simplicity of the example suppose that the variable `A` (resp. `C`) already contains a reference to the component `A` (resp. `C`), but GCMscript also defines primitives for fetching a component in the hierarchy or creating a component. The second instruction (2) corresponds to line 3 of the script. It illustrates the distributed interpretation of reconfiguration scripts enabled by our approach. It triggers the interpretation of the action `action_b` by the GCMscript interpreter of component **B**.

DOI: 10.1002/spe

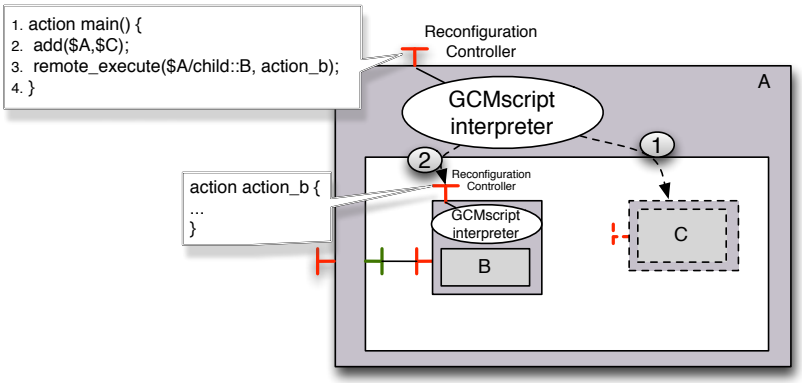


Figure 7. GCMscript interpreter: principles

Note that the first argument of `remote_execute` uses an FPath expression to access the sub-component **B**.

From the execution model point of view, reconfiguration actions are transmitted between components as requests and enqueued in the request queue of the target component. When they are served, these reconfiguration requests are interpreted by a local GCMScript interpreter embedded in the component membranes. We find this approach particularly convenient for implementing autonomic components, as illustrated below.

4.2. Autonomicity

We acknowledge that applications nowadays need to react quickly and adapt themselves to changing situations. While this concern should be included in the design of an application, it is not always clear how to embed it along the functional task of the application, as the capability for dynamic adaptation is a concern that may traverse several components of the application. It makes sense, thus, to consider the adaptation capability as a non-functional concern.

GCM/ProActive includes a mechanism that allows us to embed this behaviour in a transversal way in GCM applications. This mechanism requires the capability to reconfigure the system according to the needs. While GCMScript already allows us to modify the composition of the application in a dynamic and distributed way, this reconfiguration must be triggered by an external actor, which may be another component, or a user program. In our vision, components should also be able to take these decisions by themselves and to self-adapt to certain situations and thus, they should be made *autonomic*.

4.2.1. Autonomic Computing In order to give autonomic capability to GCM components, we resort to the *Autonomic Computing* initiative pushed by IBM as response to the ever increasing complexity in the maintenance of computing systems [62]. As the abilities required for installing, configuring, and optimizing such systems become too complex for system integrators and managers, it becomes even more difficult to deliver decisive responses in a timely way.

The vision of autonomic computing is based on the idea of self-governing systems. These systems can manage themselves given high-level objectives from an administrator [63]. In complex computing systems, management tasks like installation, configuration, protection, optimization, are not the main functional objective of most applications, but if they are not properly addressed, the application cannot accomplish its functional task. The proposition, then, is to promote self-governing systems that manage all these non-functional tasks in the background, while the application and consequently the developers can concentrate on the main functional goals. In order to effectively free developers from the burden of programming self-governing features for their applications, there must exist a way to develop these concerns independently and to integrate them with the application at a later stage.

One of the most common ways to provide autonomic behaviour is to rely on a *feedback control loop* that follows four canonical activities: *collect*, *analyse*, *decide*, and *act*. This idea evolved into a reference architecture for autonomic computing [26] in which the central element, called *autonomic manager*, implements the activities defined by the generic feedback control loop into an *autonomic control loop*. This loop defines four phases: *Monitor*, *Analyse*, *Plan*, and *Execute* and it is usually referred to as the *MAPE* autonomic control loop. The reference architecture also considers the possibility of arranging several autonomic managers in a hierarchical architecture [26], making the autonomic behaviour more scalable and supporting different autonomic control loops.

Following this view, we propose a framework for GCM components that provides autonomic behaviour through MAPE loops, and that allows these loops to interact through the hierarchical nature of GCM applications. We describe now the details of this proposition.

4.2.2. Autonomic managers for GCM components In order to effectively separate the autonomic concerns from the functional part, the autonomic behaviour must be inserted in the membrane of GCM components. Our first approach is to consider a NF *autonomic manager* component which embeds the autonomic behaviour. This component is inserted in the membrane of a GCM component and the owner of the membrane becomes, then, a managed component. The autonomic manager will observe the managed component and potentially trigger adaptation actions over it. Figure 8(a) shows a composite component **A** that has been turned autonomic by inserting an *autonomic manager* component in its membrane.

The autonomic behaviour is provided by a MAPE loop. This loop may be completely implemented by a primitive autonomic manager component, as illustrated in Figure 8(a). However, that design is not flexible enough in case we need to modify only some part of the autonomic behaviour. Instead, we choose to provide a more flexible design where each phase of the MAPE loop can be implemented in a separate way through different GCM components in the membrane. Concretely, our autonomic manager is separated in four components: Monitoring, Analysis, Planning, and Execution. This design intends to provide flexibility in two senses: (1) instead of forcing every GCM component to be fully autonomic, we allow them to implement only certain phases of the MAPE loop according to the degree of autonomic behaviour needed, and add them or remove them at runtime; and (2) by separating the MAPE loop in four components it becomes possible to modify the implementation of one specific MAPE phase without affecting the others.

Figure 8(b) shows the managed component **A** with all four MAPE components in the membrane and the additional interfaces that have been added to composite **A**. The general flow of the framework is as follows. The *Monitoring* component collects monitoring data from component **A** using the specific means that **A** may provide, e.g. using sensors for reading certain parameters, intercepting requests on the functional interfaces, or asking to the *Monitoring* interfaces of other components. Using the collected monitoring data, the *Monitoring* component computes a set of metrics and makes them available through an interface called *metrics*. The *Analysis* component provides an interface called *Rules* for receiving and storing conditions that must be verified during the lifecycle of the component. These conditions generally express desired values of some metrics. At runtime, the *Analysis* component checks those conditions using the values that it obtains from the *Monitoring* component. Whenever a condition is not fulfilled, the *Analysis* component sends a notification to the *Planning* component through an interface called *Alarm*. The *Planning* component creates an adaptation plan as a sequence of actions that can modify the state of the service and take it to a desired objective state. For taking decisions and feeding the required parameters, the planning component can obtain information from the *Monitoring* component. The sequence of actions created by the *Planning* component are sent to the *Execution* component. The *Execution* component executes the actions using the specific means that the component allows. As we envisage the possibility of having actions that can be propagated to other components, the *Execution* component is able to delegate some actions to the *Execution* component of other services through the *Actions* interface. This way, the loop is completed and the new information collected by the *Monitoring* component will reflect the new state of the application.

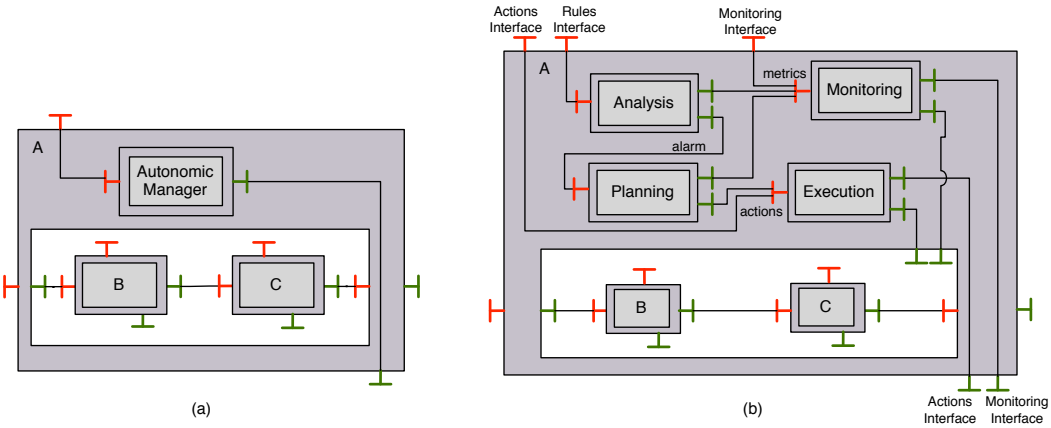


Figure 8. (a) Primitive component implementing autonomic behaviour in the membrane of composite A. (b) Autonomic behaviour implemented as a MAPE loop with one component for each phase. NF bindings to internal components have been omitted for readability.

This design has been designed as independently from the functional part as possible. However, every implementation that intends to manage a concrete application has to be adapted to the specific goals of the application and to communicate with it. Also, the way to obtain information from the managed component, or the way to trigger actions on it may also depend on the specific means that the functional implementation permits. Consequently, our design is generic for any GCM application until the point that we must define the concrete sensors and actuators that must interact with the managed component. Those sensors and actuators will be implementation dependent. We propose a possible architecture of a MAPE loop with sensors and actuators in Section 5.2.

We extended the GCM/ProActive reconfiguration API with MAPE specific aspects. The new API allows the insertion or removal of each phase of the MAPE loop at runtime; we also define a set of basic interfaces that each implementation of the MAPE loop must provide and possibly extend in order to expose the autonomic management capabilities, and to allow the communication between the MAPE loops of different components. This results in a set of NF interfaces that must be added to each membrane, see Section 5 for details on this API.

4.2.3. Interacting Autonomic Managers In many situations, autonomic managers should interact with the autonomic managers of other components. These autonomic managers may belong to subcomponents in the case of a composite, to the parent component, or to external components that are in the same hierarchical level. Figure 9 shows the interactions of the Autonomic Manager of composite **A**[†] with both subcomponents **B** and **C** using an internal NF client multicast interface. At the same time, the autonomic manager of **B** uses an external client NF interface to interact with its peer component **C**. Finally, component **C** uses its own client NF interface to reach the membrane of its parent component and communicate with the autonomic manager of **A**.

By allowing the autonomic manager to communicate with the membranes of internal components, we allow the construction of hierarchically organized autonomic managers. In such a setting, a set of low-level managed resources may have an autonomic manager that controls specific characteristics of the resource like storage space, free memory, and CPU utilization. A second-level of hierarchical managers may be in charge of higher level tasks like self-optimization, or self-healing over sets of these low-level resources. In the top of the hierarchy, an autonomic manager may have a global view and orchestrate the activity of the middle-level autonomic managers by giving priority to some autonomic behaviours over anothers, or by solving conflicts. Such a hierarchy of autonomic managers is shown on Figure 9, where the autonomic manager of **A** can control the autonomic

[†]Remember that each autonomic manager can be realised by four components constituting the MAPE loop, see Section 4.2.2.

DOI: 10.1002/spe

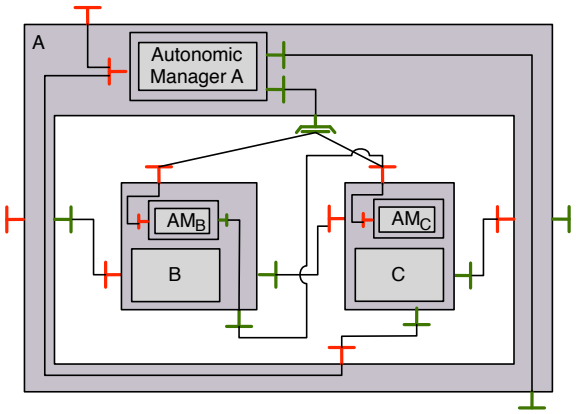


Figure 9. Composite GCM component with Autonomic Management bindings. Autonomic Manager of **A** is bound to Autonomic Managers of **B** and **C**.

managers of components **B** and **C**. Another setting may consider several autonomic managers collaborating as peers, at the same hierarchical level; for example to perform load balancing, or by performing coordination tasks. Both kind of architectures can be built using GCM autonomic components.

4.2.4. Autonomic Adaptation The reconfiguration language presented in Section 4.1 provides a convenient approach for programming the execution part of the MAPE loop. Our asynchronous components equipped with MAPE loops programmed thanks to GCMScript provide a powerful framework for easily programming components as independent entities, both from the point of view of the execution flow, thanks to a request/reply mechanism, and from the point of view of the component management, thanks to componentized membranes equipped with MAPE tools. This way, both the business code and its management are organized in a componentized manner, where components act as loosely-coupled entities.

Section 6 illustrates the integration of all those notions in a concrete scenario, showing that our framework is particularly convenient for programming autonomically adaptable components, and that our autonomic managers are able to delegate adaptation decisions to sub-components, or to external components. This scenario leverages the GCM-based MAPE framework and related APIs.

5. IMPLEMENTATION

The purpose of this section is to discuss aspects related to the implementation of GCM components. The first part of this section focuses on the implementation of GCM/ProActive and its autonomic capabilities. Then, we describe the integration of GCM/ProActive components into SCA assemblies. Our aim in this last part is to show that the GCM/ProActive framework is generic enough to integrate smoothly with other component models like SCA, but also that our component model is indeed adapted to the integration within a service-oriented environment.

5.1. GCM Implementation

GCM/ProActive is an implementation of GCM over the ProActive middleware. ProActive is a Java middleware that provides an API to instantiate and run active objects; it implements the mechanisms used by our asynchronous components, like request queues and futures. Whereas the basic programming features like futures and request service is provided transparently to the programmer, additional non-functional features of active objects are accessible through the ProActive API (for instance to migrate a component or to check if a request has been served). GCM

components are implemented in ProActive using active objects in such a way that each component is implemented by exactly one active object, and serves requests as described in Section 3.4.

GCM/ProActive provides an API to instantiate components. Components are instantiated by defining a *ComponentType* object that specifies the set of F and NF *InterfaceType* objects that will be exposed on the membrane of the component. An *InterfaceType* object contains the attributes of a single component interface: name, signature (a Java Interface), role (client or server), contingency (mandatory or optional), and cardinality (singleton, multicast, gathercast). An *Interface* in GCM/ProActive is defined by an *InterfaceType* and an actual implementation. The implementation of an *Interface* is any object that provides the signature and may be initially undefined and be assigned at runtime using the component binding mechanism ; in that case, its contingency value must be "optional".

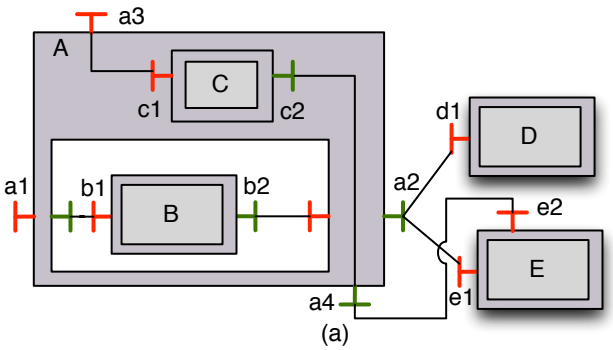
GCM/ProActive components may also be defined in a simpler way by using a GCM-ADL file that is compiled by an ADL Factory, which instantiates the application. A GCM-ADL file is an XML descriptor that extends Fractal ADL to allow the definition of collective interfaces, and of NF components in the membrane. The composition of the membrane follows the same rules as any other composition, and may be defined either inside the `<controller>` tags or in a separate GCM-ADL file specified as an attribute of the `<controller>` tag.

Figure 10 shows a simple example of a GCM/ProActive application and a part of the instantiation process. Figure 10 (b) shows the instantiation of component **A** through the GCM API by defining their *InterfaceType* and *ComponentType* objects, and then using the *MembraneController* to insert component **C** inside the membrane of **A**. Prior to initiate the lifecycle of **A**, its membrane must be started. Figure 10 (c) shows the ADL for the instantiation of the entire application of Figure 10 (a); it consists of the main file *GCMApp.fractal*, and the additional file *NFCompA.fractal* that defines the membrane of component **A**. The architectural definition using the GCM-ADL allows for more concise descriptions in terms of coding effort and is less error-prone as the ADL Factory translates the descriptions to the appropriate GCM API invocations. Also, the ADL description eliminates the need to recompile the application code for creating a different component assembly, thus facilitating its manipulation by external tools. Being able to specify the F and NF parts using two separate ADLs enforces the separation of concerns that we advocate in the design of GCM applications; the two ADLs are merged at instantiation time. Finally, Figure 10 (d) shows the only two lines of application code required to instantiate the application, which is much shorter compared to Figure 10 (b).

Deployment of functional and non-functional components is handled by the ProActive middleware. ProActive enforces a separation between the programming of the distributed application and the actual deployment by using a Virtual Node (VN) abstraction [1]. GCM components are associated to a VN, either using the API or using the GCM-ADL file. When actual deployment takes place, GCM/ProActive uses an underlying mechanism that allows us to map the specified VNs to an actual infrastructure. The mechanism to access the infrastructure ranges from simple access to remote machines using *ssh*, to complex resources managers and job schedulers. Once the remote resources are acquired, a set of JVMs are instantiated on the VNs, according to the mapping mentioned above. The active objects that implement the GCM/ProActive components are hosted on these JVMs. This separation between the application description, specified in terms of VNs, and the mapping to the actual infrastructure, allows any ProActive based application to be deployed in a variety of infrastructures like clusters, grids or clouds, without changing or recompiling the application.

5.2. MAPE Framework in GCM/ProActive

GCM/ProActive provides a basic implementation of the MAPE loop as described in Section 4.2. This implementation provides an API for each phase of the MAPE loop and a working implementation of one component for each phase. The insertion of components in the membrane is done by a provided console application which is no more than a front-end that uses the NF API of GCM to introspect the application and automatically insert or remove the MAPE components as needed. The API for the MAPE loop is shown in Figure 11.



(a)

```
// composite component 'A':
PAGCMInterfaceType[] fTypeA = new PAGCMInterfaceType[] {
       typeFactory.createGCMItfType("a1", "A1Itf", SERVER, MANDATORY, SINGLETON),
       typeFactory.createGCMItfType("a2", "A2Itf", CLIENT, OPTIONAL, MULTICAST) };
PAGCMInterfaceType[] nfTypeA = new PAGCMInterfaceType[] {
       typeFactory.createGCMItfType("a3", "A3Itf", SERVER, OPTIONAL, SINGLETON),
       typeFactory.createGCMItfType("a4", "A4Itf", CLIENT, OPTIONAL, SINGLETON) };
ComponentType compAType = createFcType(fTypeA, nfTypeA);
Component A = gcmFactory.newFcInstance(compAType, new ContentDescription(null),
       new ControllerDescription("A", COMPOSITE, controllerConfigurationFile), node);
Component B = // ... definition for F component B
Component C = // ... definition for NF component C
GCM.getMembraneController(A).nfAddFcSubComponent(C); // insertion of C into membrane of A
// ... definition of bindings inside the membrane of A
GCM.getContentController(A).addFcSubComponent(B); // insertion of B into composite A
// ... definition of bindings inside functional part of A
GCM.getMembraneController(A).startMembrane();  // start of membrane lifecycle
GCM.getLifecycleController(A).startFc();  // start of lifecycle of A
```

(b)

```
// file: GCMApp.adl
<definition name="GCMApp">
  <component name="A">
    <interface name="a1" signature="A1Itf" role="server" contingency="mandatory" cardinality="singleton"/>
    <interface name="a2" signature="A2Itf" role="client" contingency="optional" cardinality="multicast"/>
    <component name="B">
      // definition for component B
    </component>
    <controller desc="NFCompA"/> // File "NFCompA.adl" describes the NF part
    <binding client="this.a1" server="B.b1"/>
    <binding client="B.b2" server="this.a2"/>
  </component>
  <component name="D">
    // definition for component D
  </component>
  <component name="E">
    // definition for component E
  </component>
  <controller>
    <binding client="A.a4" server="E.e2"/>
  </controller>
  <binding client="A.a2" server="D.d1"/>
  <binding client="A.a2" server="E.e1"/>
</definition>
```

```
// file: NFCompA.adl
<definition name="NFCompA">
      <interface name="a3" signature="A3Itf" role="server" contingency="optional" cardinality="singleton" />
      <interface name="a4" signature="A4Itf" role="client" contingency="optional" cardinality="singleton" />
      <component name ="C">
        // definition for component C
      </component>
      <binding client="this.a3" server="C.c1"/>
      <binding client="C.c2" server="this.a4"/>
</definition>
```

(c)

```
// instantiation of application from ADL file
String adlFile = "GCMApp.adl"
Component gcmApp = ADLFactory.newComponent(adlFile, context);
```

(d)

Figure 10. (a) A sample GCM application; (b) Partial instantiation of composite A using the GCM-API; (c) Instantiation of the application using two GCM-ADL files; (d) Instantiation using GCM-ADL

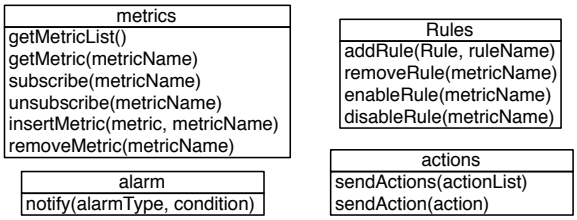


Figure 11. Basic API for the MAPE components

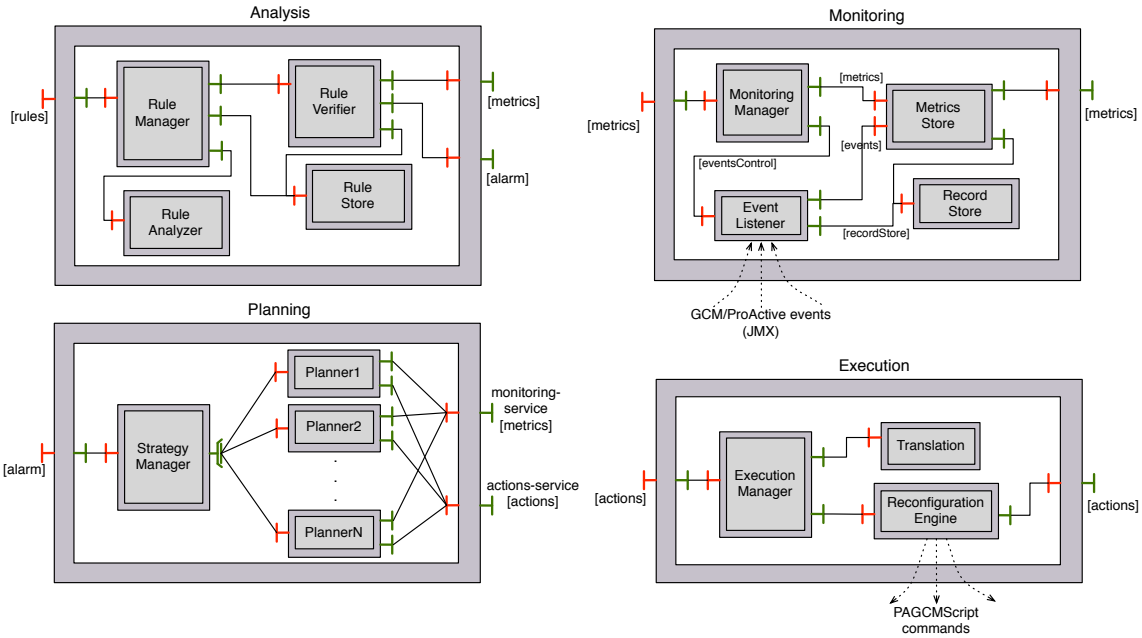


Figure 12. Internal composition of the MAPE components

The internal composition of the MAPE components in GCM/ProActive is shown in Figure 12. This set of predefined components that implement each phase is just one of several possible implementations that illustrates well how to provide adaptability.

In this implementation, the *Monitoring* component receives a Metric that is a Java object with a *compute* method, and inserts it in the *Metric Store*. The *Metric Store* provides the connection to the sources necessary for the computation of the Metrics; namely, the *Record Store* to get already sensed information, the *Event Listener* to receive sensed information directly, or the *Monitoring* component of other components, allowing access to the distributed set of monitors. The *Event Listener* collects JMX events that are produced by the ProActive middleware. These events include the sending, starting, and finishing time of requests, as well as the update of futures. The *Record Store* stores records of monitored data that can be used for later analysis. To gather external information, we have also designed a set of JMX probes for CPU load and memory use, and incorporated them along with the events produced by ProActive within the store.

The *Analyzer* component queries the *Monitoring* component. The *Analyzer* consists of (1) a *Rules Analyzer*, which transforms the Rule description to a common internal representation, (2) a *Rule Store* that maintains the list of Rules, (3) a *Rules Verifier* that collects the required information from the *Monitoring* interface and generates alarms, and (4) a *Rules Manager* that manages all the process. A Rule is described as a triple $\langle metricName, comparator, value \rangle$, where *metricName* is the name of a metric. The Rule Verifier subscribes to the *metricName* from the *Monitoring* component (either in a push or in a pull mode) to get the updated values and check the compliance of the Rule.

The *Planning* component includes a *Strategy Manager* that receives an alarm message and, depending on the content of the alarm, triggers one *Planner* component, among a set of bound *Planners*. Each one of the *Planner* components implements a planning algorithm that can create a plan to modify the state of the application. Each *Planner* component can access the *Monitoring* components to retrieve any additional information they may need; the output is expressed as a list of actions in a predefined language.

In our implementation we profit from the capability to dynamically select one of the bound interfaces of a one-to-many (multicast) interface provided by GCM to decide which *Planner* component will be triggered. For example, if the rule violated is related to response time, we may trigger a planner that generates a performance-oriented recomposition; or if a given cost has been surpassed, we may trigger a cost-saving algorithm. The decision of what planner to use is taken in the *Strategy Manager* component. Nevertheless, the possibility of having multiple strategies might be a source for conflicting decisions; while we do not provide a method to solve these kinds of conflicts, we assume that the conflict resolution, if required, is provided by the *Strategy Manager*.

As an example, we have implemented a simple planning strategy that, given a particular request r, computes the *requestPath* of r, i.e. the list of sub-requests that were called during the service of r along with the time spent by each sub-request. The strategy finds the component most likely responsible for having broken the Rule that triggered the alarm; this can be done examining the result of the *requestPath* computation and determining the component c that took the largest time serving the sub-requests. The strategy then creates a plan that replaces c for another component from a set of possible candidates. More complex strategies could be designed, based on additional information from the application, the execution environment (e.g., available resources), etc.

The *Execution* component receives actions from the *Planning* component. It includes an *Execution Manager* that manages the execution , and a *Reconfiguration Engine* that embeds a GCMScript interpreter. As reconfiguration actions triggered by the planners may be expressed in different formats, the *Execution Manager* may use a *Translation* component to translate actions before transferring them to the *Reconfiguration engine*. The *Execution Manager* may also discriminate between the actions that can be executed by the local component and those that must be delegated to external *Execution* components.

This framework exemplifies how NF components can be used in GCM/ProActive to provide autonomic capabilities through MAPE loops. By providing specific implementations of each phase of the loop while adhering to the interaction means presented it is possible to provide other, more complex autonomic features without modifying the functional part of the GCM application. Below, we describe our efforts to provide another level of flexibility by showing how, using GCM/ProActive features, it is possible to build GCM based applications interacting with applications based on SCA.

5.3. SCA interoperability

The purpose of this section is to describe how GCM/ProActive component applications can be integrated within a service-oriented environment, and more particularly an SCA assembly. We think that this contribution is crucial as it shows that our programming and composition model is indeed convenient for effectively building large-scale service-oriented applications. GCM/ProActive framework is not only an effective programming model, it is also generic and open and allows for the integration with other different composition models.

Our overall goal is to provide GCM components with the possiblity to be integrated within SCA component based applications. I.e., our objective is to be able to expose GCM components as SCA ones, and to compose applications made of GCM components and other SCA components. Typically, services requiring high performance computing and high scalability, through parallel execution on Cloud resources will be instantiated as GCM/ProActive components, and a SOA application will be composed from those high performance components and other SCA components provided by third parties and running on other SCA platforms.

To this aim, GCM components must feature additional aspects to be able to be integrated as specific parts within an SCA application. Among them, the inter-components communication mechanism is key, and we have chosen Web Services as a communication protocol to support SCA

bindings, it is the most standard communication mechanism used for SOA applications. Besides, SCA specification includes SCA intents and an ADL-driven approach to describe component assemblies. We also made GCM/ProActive compliant to those aspects in order to allow for a full integration of GCM components into SCA applications. Notice that our goal is not to implement a full-fledged SCA platform on top of GCM/ProActive, as FraSCAti [11] for instance, but it is rather to offer mechanisms that can ease the integration of GCM asynchronous components, with their peculiar features, as SCA entities able to connect with third parties SCA components. Overall, our objective is to be able to extend SCA applications with the possibility to integrate high-performing, large-scale distributed, and autonomous components.

5.3.1. Web service based component interaction As any ProActive active objects, GCM component interfaces can be annotated as Web Service supported communication channels. A specific component controller is available for selecting which server interface of the component to expose. A server interface exposed this way translates into the automatic deployment of a web service[‡] (WS) at runtime. The role of this WS is simply to delegate an incoming method call on this interface to the effective GCM server interface object, automatically blocking on the received future in case this method intends to return a value, and sending back this value to the caller acting as the WS client. The corresponding WS endpoint reference serves as the information to be stored at the client side when binding an interface to this server interface.

Because invoking a web service method is blocking for the caller until the reply gets back, a GCM client interface which is bound to a WS exposed server interface is automatically attached a proxy which replaces the default ProActive proxy. The role of this proxy is to trigger the WS invocation, wait for the reply, and only then, return this reply to the component code. As the method call is not based on ProActive requests, it is not possible to return a future reference to the caller, because it is impossible to support its later assignement. As using ProActive requests is transparent anyway, replacing the proxy that handles such requests by a proxy that plays the role of a usual WS client has no impact on the component code. Also, the implementation provided by default for this proxy can be easily replaced by an ad-hoc written one (e.g. if deciding to connect to a WS-Rest instead of WS-Soap service implementation): which proxy implementation class to use is configurable through a specific syntactical construct of the GCM ADL.

5.3.2. SCA intents As specified by the SCA policy [64], each method offered or required by any SCA component interface can be augmented with some intents. A typical intent can ensure security of service invocation, by cyphering, authenticating, or guaranteeing integrity of the exchanged messages. Note that these intents are usually involving the caller and the callee of a method. This is why GCM/ProActive interoperating with full-fledged third-party SCA components must support SCA intents even if our objective is not to provide a full-fledged SCA compliant framework.

Through policies, specific implementations of intents are made available, and consequently, the user composing the application is offered the possibility to attach any intent to any interface method by indicating which policy is chosen. The order in which the intents are selected dictates the order in which they will be applied. We added to GCM/ProActive a new NF controller acting as a SCA intent controller; it offers methods to add, remove, and query intents relatively to a method of a functional interface of the component. Notice that similarly to FraSCAti [11] and beyond the SCA specification which only deals with design time, we support the possibility to dynamically modify the list of intents attached to a given method.

Internally, to attach an intent to a specific method of a specific interface, we must intercept incoming and outgoing requests in order to realize the specified intents. For this, we would need GCM/ProActive to support the notion of input and output interceptors. An interceptor could be seen as a component in the membrane that would be directly bound to (external and internal) functional interfaces that it connects (see Figure 1); in Fractal, it is possible to have an interceptor object in the membrane. However, the precise design of such controllers as components is trickier than for other

[‡]implemented thanks to the Apache CXF Framework `cxf.apache.org/`

NF components: interceptor components deal with non-functional aspects (like security) but are mainly bound to functional interfaces, and should have a restricted impact on the flow of functional requests. As the design of interceptors as components is a complex issue, they are not yet part of the GCM model, and consequently we could not use them here. Extending GCM to support component interceptors is outside of the purpose of this article.

Instead, to support dynamic intents in Java-based GCM components, we dynamically generate the bytecode corresponding to a subclass of the one implementing the functional part of the component (using Javassist[§]) . In this subclass, each method is redefined in such a way that it will sequentially run a list of additional codes corresponding to the currently deployed intents, both at entry and at exit of the original method call. So far, to prove the feasibility of our approach, we have only implemented intents for security (using pairs of public/private keys provided at intent deployment time). As we do not implement the PolicySet feature of the SCA intent specification, we didn't develop elaborated mechanisms to group alternate intent codes as sets in codebases.

5.3.3. SCA component instantiation Our goal is also to allow a component described as an SCA component to be instantiated as a GCM one. GCM and SCA share many features regarding the initial description of the component assembly forming an application. Besides the description of the SCA assembly using an XML dialect as we will discuss later, for Java components SCA builds upon annotations within business code to make the component oriented features of such code explicit. At instantiation time, the code of an SCA component is introspected in order to detect the presence of these annotations and react accordingly. To instantiate a GCM component based on SCA annotations, we need to generate code specific to the GCM component model. Next table summarizes the set of basic SCA features and corresponding annotation, and their translations in the GCM platform.

	SCA	GCM
Property injection	@Property	AttributeController
Component name	@ComponentName	NameController
Required/Offered Service	@Service/@Reference	Interface Server/Client

For instance, if the @reference SCA annotation is attached to a field declaration in the business class, this means this field stores the reference of the bound component. Consequently, a subclass of the original Java implementation class is generated using Javassist, which adds all GCM API features related to the management of a binding within the code: all methods of the GCM/Fractal BindingController NF interface are implemented considering the field name as the bound component reference. If the annotation regards a security intent for instance, the code for holding the private/public key pair is also generated in the corresponding subclass that is also required for applying the additional needed intent code as explained previously.

Concerning the SCA ADL language, it exhibits XML elements constructs that can easily be translated into the corresponding GCM ADL format. To this aim, we have simply added a front-end module to the GCM ADL factory: given an SCA ADL file, this module translates this file into a GCM ADL one and will proceed with the normal process of instantiating the resulting GCM component. Notice that the presence of the GCM controllers allows the introspection and the reconfiguration of the component, which is originally not supported by SCA. Furthermore, this module automatically annotates each SCA server interface to be exposed as a web service, and adds specific SCA controllers: for handling SCA binding types, SCA properties and SCA intents. When instantiating primitive component, the factory additionally triggers the Javassist code generation mentioned so to handle SCA annotations.

Consequently, specific GCM features requiring to be exposed through the corresponding GCM ADL file can not be supported. For example, as SCA ADL does not support non-functional interfaces, nor collective interfaces, it is impossible for our SCA extended GCM ADL factory to parse an SCA component description that will directly turn into a GCM component exhibiting such

[§]Javassist www.csg.is.titech.ac.jp/~chiba/javassist/tutorial

kind of interfaces. Furthermore, Fractal and GCM impose that all interfaces forming the component type even if optionnally implemented, are described at component instantiation time; it is thus impossible to add those GCM specific interfaces dynamically. However, it seems possible, in the future, to extend GCM in order to support dynamic interface creation and thus eventually obtain full-fledged GCM high-performance autonomic components generated from SCA ADL assemblies and SCA annotated business code. An alternative and already practical solution is to simply allow the user to manually add GCM ADL lines in the automatically generated GCM ADL file for the required specific GCM features, before proceeding to the normal instantiation process.

This section showed how to take full advantage of GCM components in the scope of third party SCA services. The presented extension to GCM/ProActive enables for instance a typical integration of services to compose an SOA application as follows: have third party SCA components, hosted by any SCA framework (e.g., Tuscany, FraSCAti), bound to a SCA/GCM "front-end" component exposing its interfaces to the outside as web services. This SCA/GCM component is acting as an entry/exit point of an SCA domain; it gets bound, through SCA references implemented as GCM bindings, to some independently instantiated GCM components. Those GCM/ProActive components, created by a GCM ADL, are able to feature self adaptation and high performance thanks to the use of GCM/ProActive asynchronous components. All these components, especially the "front-end" one, can even be designed as SCA components eventually enriched with GCM distinctive features, and further instantiated as GCM/ProActive ones. Such a pattern can be useful whenever one specific service within the SOA application has to fulfill service level objectives regarding e.g. performance level, hosting cost, etc. Such constraints may indeed require that the service itself is internally parallelized and distributed e.g. on a private cluster, a public Cloud, etc., and is self-adaptable regarding its use of other services [65].

6. USE CASE: AUTONOMIC RECONFIGURATION

GCM/ProActive has been used to implement several use-cases as real-world applications, some of them being briefly cited in the conclusion of this paper; each of those use-cases benefit from some of the capabilities of our component model. The purpose of this section is so to illustrate, thanks to a specifically built scenario, the integration of all the previously presented notions at the same time. We show below that our framework is not only particularly convenient for programming autonomically adaptable components, but also effective in practice. More precisely, we describe a system performing some autonomic adaptation actions, able to also rely on sub-components or external components to take its adaptation decisions.

Compared to some of the related works (e.g. autonomous home control system as an illustrative use case of FraSCAti [11]), the fact that the GCM/ProActive implementation includes a MAPE compliant framework considerably drives the autonomicity feature programming in practice. Other frameworks, also inspired by the MAPE model exist and target similar goals (Behavioural Skeletons [66], Dynaco [39] for any sort of application, Safdis [67] for service-oriented applications). However, none pushed the efforts as far as we did, including from an engineering perspective, to demonstrate that it is of practical interest to cleany and entirely separate the concern of autonomicity management from the functional code.

6.1. Use case description

Consider a GCM/ProActive application as shown in Figure 13. Component **G** provides a GUI that acts as a front-end for a user to submit computations to a farm of worker components contained in the composite **F**. The composite contains three *worker* components **W1**, **W2**, and **W3**. These workers encapsulate real or virtual instances that perform a computation and store their results in a common repository, provided by the component **R**.

In this setting, each *worker* has a Monitoring component in its membrane. Each Monitoring component implements machine-dependent metrics like *freeDisk* space, *freeMemory*, and *idleCPU*

DOI: 10.1002/spe

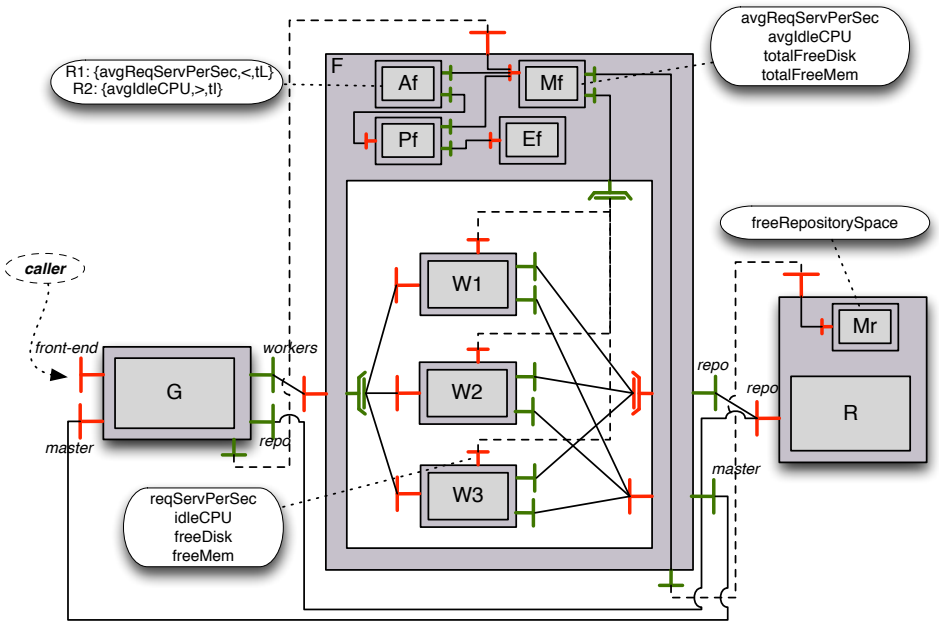


Figure 13. GCM autonomic application example. A farm of workers that monitors its load and can adjust the number of workers.

```
action addWorker() {
  $W4 = newComponent("worker.adl");
  add($F,$W4);
  bind($F/interface::wi,$W4/interface::wi)
  bind($F/interface::wo,$W4/interface::wo)
  bind($F/interface::wm,$W4/interfaces::master)
  bind($F/interface::intmetrics,$W4/interface::metrics)
}
```

(a)

```
action changeRepository(RP) {
  unbind($F/interface::repo);
  unbind($F/interface::extmetrics);
  unbind($G/interface::repo);
  bind($F/interface::extmetrics,$RP/interface::metrics)
  bind($F/interface::repo,$RP/interface::repo)
  bind($G/interface::repo,$RP/interface::repo)
}
```

(b)

Figure 14. GCMScript sample codes. (a) Adds a worker to the farm. (b) Changes external repository to **RP**

time, and also provides a logical metric *reqServPerSec* that counts the number of requests that have been served per second by this *worker*.

The autonomic behaviour is provided by a MAPE loop in the membrane of **F**. In this loop, the Monitoring component $\mathbf{M}_f$ collects the metrics of each worker using NF bindings to each *metrics* interface, shown as dashed lines in Figure 13. This way, $\mathbf{M}_f$ can compute an aggregated *avgReqServPerSec* metric for the system and, at the same time, be aware of the machine status of each worker. The Analysis component $\mathbf{A}_f$ contains two rules that intend to regulate the load of the set of workers. The first one triggers an alarm when the amount of operations per second is lower than the threshold t_L. The second one triggers an alarm when the average idle time of the workers is bigger than t_I. The Planning component $\mathbf{P}_f$ reacts according to the kind of alarm received. In the case of the first alarm, $\mathbf{P}_f$ checks the amount of idle time of the workers to confirm that they are all busy most of the time and diagnoses that the low number of operations per second is due to overload of the workers, so it creates a script (in GCMScript) that allocates a new worker inside **F** and creates the bindings to add it to the farm. In the case of the second alarm, $\mathbf{P}_f$ diagnoses that the workers are being sub-utilised and decides to remove one, so it chooses the worker that has had the least load in the last minute and creates a script that removes that worker from $\mathbf{P}_f$, unless there is only one worker inside, in which case it does nothing. The Execution component $\mathbf{E}_f$ is the interpreter that embeds the GCMScript engine and is able to execute the instructions over the composition. Figure 14(a) shows an example of a GCMScript code that adds a new *worker* to the farm.

The metrics provided by $\mathbf{M}_f$ are accesible by the membrane of component **G** which can poll it and use it to display to the user the current status of the farm. **G** does not have an autonomic behaviour, as it just implements a Monitoring component $\mathbf{M}_g$ that collects data from $\mathbf{M}_f$. Component **R** is also non-autonomic and it implements a Monitoring component $\mathbf{M}_r$ that checks the amount of free space on the repository. This metric, called *freeRepositorySpace* is exposed by $\mathbf{M}_r$ and read by $\mathbf{M}_f$. Component $\mathbf{A}_f$ may include another rule that is activated when the *freeRepositorySpace* reaches a low threshold, and sends an alarm to $\mathbf{P}_f$ that may decide to unbind **R** and use another equivalent repository **RP**. The sample code is in Figure 14(b).

This way the MAPE loop of **F** monitors not only the internal behaviour of the workers, but is also able to check the state of the repository. It is important to note that the component based approach to the MAPE loop implementation allows the modification of several parameters of the autonomic loop, for example, one can adjust the thresholds in $\mathbf{A}_f$ or changing the strategy in $\mathbf{P}_f$ without affecting the behaviour of $\mathbf{M}_f$ or $\mathbf{E}_f$.

6.2. Experimental results

Experimentation was done over an implementation of the example described in Section 6.1, where component **F** is used as a blocked matrix multiplication solver that distributes a set of tasks to the workers. The solver takes as input matrices of size $2^N \times 2^N$, and generates $2^{2(N-b)}$ tasks to distribute on the workers, where b means a 2^b block size for the solver. Thus, for a fixed b, the number of tasks grows exponentially on N.

An extract of the implementation code is shown in Figure 15. Component **G** receives computation requests from an external caller through the *front-end* interface. **G** acts as the master of the computation, creates a set of tasks to solve, and then starts all workers **W** using a multicast call. Once started, workers asynchronously poll the repository of tasks to get one task through their *master* interface, solve it, send the results to **R** through their *repo* interface and reenter a request in their own queue unless there are no more tasks. This behaviour is easily implemented using multicast interfaces and asynchronous invocations. Note that, once workers have started, component **G** can terminate the service of method `multiply()` and is free to accept new multiplication requests that will be added to the tasks repository. In a synchronous setting, component **G** would block inside the `multiply()` method until all workers have finished their tasks. Using asynchronous calls, the assignation of tasks is implicitly triggered each time a worker requests a new one.

Results for a computation identified by `id` are stored in component **R**. The caller can retrieve the results through **G** which calls `getResult(id)` from **R**. Again, in a synchronous setting, this call would block until all tasks have been computed. However, in this case the method returns immediately; if all tasks for the matrix `id` have been computed, the actual result is obtained; otherwise, a future object is transparently sent to the caller. Upon examination of the result, the caller outside **G** may block, but **G** can continue its execution.

Our baseline is the execution of the application with an initial number of workers W and without any part of the MAPE loop included in the membrane of any component. Then we compare against a version with monitoring components included in each membrane, as well as the analysis component $\mathbf{A}_f$ in the membrane of **F** (*MA* version). This version provides information on the execution of the farm. In particular we insert the *reqServPerSec* metric in every worker, and the *avgReqServPerSec* metric in **F**. Component $\mathbf{A}_f$ includes the rule $\langle$ *avgReqServPerSec*, $<$, t_{Tr} $\rangle$, where t_{Tr} is a given threshold. In case an alarm is triggered, $\mathbf{A}_f$ notifies the user, and no other action is taken. The third version includes a $\mathbf{P}_f$ and $\mathbf{E}_f$, rendering component **F** autonomic (*MAPE* version). In this case, when an alarm is triggered, the planner $\mathbf{P}_f$ adds another worker component to the farm, using the GCMScript code shown in Figure 14.

Figure 16 shows the execution time for an initial number of workers $W = \{1, 2, 4, 8\}$ and different sizes of matrices. The version with monitoring and analysis components (MA) behaves slower than the baseline version, which is expected as we are introducing the overhead of reading metrics and checking rules but not taking any action in regard. The overhead reaches 25% in a case where we stress the monitoring and analysis components by frequently checking the stored rules. The

DOI: 10.1002/spe

```
//Extract of component G implementation class
Queue<MMultTask> tasks = new LinkedBlockingQueue<MMultTask>();

public void multiply(int[][] matrix, int b, int id) {
  // Init current set of tasks
  createTasks(matrix, b, id);
  // Workers are started using 'broadcast' mode for multicast
  workers.work();
}

public void createTasks(int[][] matrix, int b, int id) {
  N = matrix.length; B = Math.pow(2,b);
  // Create list of multiplication tasks
  tasks = new LinkedBlockingQueue<MMultTask>();
  for(int i = 0; i < N/B; i++)
    for(int j = 0; j < N/B; j++)
      tasks.add(new MMultTask(i*B, j*B));
}

public MMultTask getTask() {
  return tasks.isEmpty() ? new NullTask() : tasks.poll();
}

public MMultResult getResult(id) {
  return repo.getResult(id);
}
```

```
//Extract of Worker implementation class
public void work() {
  MMultTask task = master.getTask();
  if(task.isNullTask())
    return;
  // Execute multiplication task
  task.multiply();
  // Store result in external component r
  repo.storeResult(task);
  self.work();
}
```

Figure 15. Implementation code extracts for component **G** and a worker **W**i. **G** creates a bag of tasks, while workers asynchronously poll the bag and store results in **R**.

autonomic version of **F** adds one worker when it finds that the average number of served requests reaches the low threshold. For small sizes of the matrix and, consequently, short execution times, performance is worse than the baseline version as we must stop the activity of component **F** before adding the new worker and this reconfiguration time is too long to show a benefit. For bigger sizes, the execution time is long enough and the addition of new worker compensates the overhead of measuring and reconfiguring the application.

This example shows how the autonomic features of GCM/ProActive can be used to build an application that autonomically adapts its architecture according to certain rules by modifying bindings and compositions. However, not only bindings and compositions can be dynamically modified. In GCM/ProActive active objects and components can migrate transparently between machines that have been acquired from grid or cloud infrastructures. It is possible, then, to define migration actions as GCMScript methods and enrich the set of adaptations that can be executed as part of the autonomic loop. For example, it is possible to define a rule that checks if the *freeMem* metric of a *worker* reaches a lower bound; in case it does, a planner creates a GCMScript that acquires a new machine with higher total memory, stops the *worker*, migrates the worker to the new machine and restarts its lifecycle. From the point of view of the functional execution, the application

DOI: 10.1002/spe

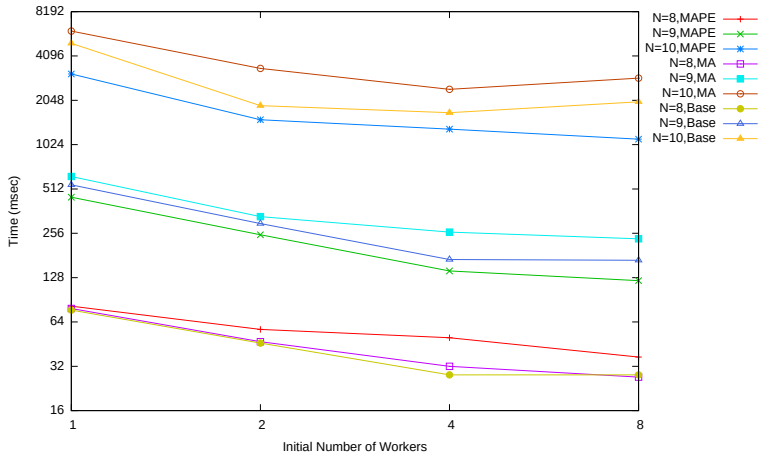


Figure 16. Execution of a matrix multiplication solver using the application of Figure 13 for three configurations: base (no component in the membrane), MA (Monitoring and Analysis activities), and MAPE (fully autonomic). For N big enough, the overhead of executing the autonomic loop is compensated by the addition of a new worker.

does not need to be aware that a component has been migrated. This feature provides another level of flexibility to the autonomic control loop.

7. COMPARISON WITH RELATED WORKS

In this section, we present a comparison of our approach with some similar approaches, either in the domain of adaptive component systems, or of distributed programming models.

High-level component models for adaptive distributed systems

First, several frameworks already exist for enabling better adaptation of Fractal components. Among those the closest work to ours is AOKell and FScript. Compared to those works, we think that our approach is better adapted to simply program autonomous and distributed components. Indeed, without our extension of FScript, distributed execution of reconfiguration script would have to be triggered manually, and the programmer would have to instantiate himself several distributed FScript interpreters. Compared to AOKell, we have a stronger execution model which is natively distributed and autonomous, it is also smoothly integrated with features of the component model. Of course, a GCM/ProActive application could be programmed relying on aspects and Fractal basic implementation (comprising FractalRMI); but this would require significant additional design and coding efforts to end up with a solution comparable to ours.

Some high-level component models targetting adaptation for distributed systems have been recently proposed. They introduce original approaches for expressing the needed dynamic reconfigurations, instead of letting the adaptation loop programmer explicitly devises them.

One of them is Kevoree (`kevoree.org` [68]). It leverages a model-driven approach, and gains its originality by relying upon such a model even at runtime. In the model, runtime entities are explicited, like nodes, components, channels, or group of components that are subject to a same synchronization policy. Kevoree approach is peculiar compared to classical ones, as the dynamic adaptation required for any runtime entity to apply is implicit: when a distributed entity receives a model update that reflects the target running software architecture, it extracts the reconfigurations that affect it and transform them into a set of platform reconfiguration primitives.

Another is DEECo [69] that targets open-ended (e.g. ubiquitous) very dynamic distributed systems. Instead of describing explicitly an architecture involving components, the model relies upon the notion of an ensemble of components which is indeed a dynamic group of components,

mutually cooperating. Bindings of components within an ensemble are not made explicitly. Implicit bindings are inferred through the knowledge about exposed or required features, and also about their reaction and adaptation abilities. The membership of a component within any ensemble is conditioned to a regularly evaluated membership condition: a given component, given its internal state, could fulfill many ensenble membership conditions at once.

Overall, Kevoree and DEECo are to be considered as higher-level adaptive and distributed component oriented models, than GCM/ProActive is. Say another way, it looks natural and more or less technically feasible to implement each of such models as a specific GCM/ProActive application strongly relying upon the provided open MAPE framework for expressing the self-adaptation policies peculiar to each of such models. The only point that sounds difficult to handle comes from the absence of shared components in GCM (compared to Fractal) as would be required to allow a given component to belong to several ensemble components of DEECo: it was a deliberate choice not to provide this concept in the GCM specification [1], not breaking the natural rule that a composite component is deemed the only manager of the distributed components it recursively embeds (and as such needs to maintain a state-full view about its content). Notice that ensemble components of DEECo are stateless.

Programming models for distributed systems

Quite a lot of successful programming models exist for programming distributed systems [70]; among them actors and active objects have recently gained interest and revealed very convenient way to program active objects. Outside the ProActive world, one can exhibit the recent successes of the actors, like in Scala [71] and Akka [72], or of the active objects, like in ABS [73] and JCobox [74]. From another point of view, several component models exist to help the programmer design large-scale adaptable distributed systems, like the ones [69, 68] mentioned above. The main originality of our approach is to merge the two worlds: we benefit from a programming model adapted to distributed computing, based on active objects, and use it to implement a component model adapted to large-scale deployment and adaptations.

Compared to other programming and runtime environment, we chose ProActive because it has a good support for deployment, and it features active-objects which is a programming abstraction more adapted to component-oriented programming than pure actors. Indeed, futures provide transparent communications of method results that are more efficient than explicit callbacks of actor models. Also, the pure actor paradigm consists in reacting to incoming messages and possibly changing the actor behaviour, while active objects serve incoming remote method invocations and possibly change their internal state. Those two differences make the active-object programming model particularly close to the usual abstractions of object-oriented programming, which, to our mind, is closer to component-oriented programming than the functional programming model encouraged by actors. However an effective implementation of GCM could also be realised based on any other actor or active-object programming language.

Another related research area is the work on connectors and their automatic generation [75]. In those works, bindings between components are automatically equipped with additional (mostly non-functional features). Compared to our approach, the objective is similar; however, we chose to encapsulate non-functional features at the level of the component, inside its membrane, potentially as interceptors modifying communications.

First, encapsulating this logic at the level of the component seems more coherent with our approach where the component is the unit of distribution and of decision. One could argue that sometimes the connection could be the right level where adaptation can occur; in GCM/ProActive we chose to rely on proxy adaptation to handle such cases, see Section 5.3.1.

The second and more important reason why we do not allow complex adaptable connectors – like Fractal's binding components and their use in the Dream framework – is that we want the bindings to encapsulate a simple and uniform semantics. This paper advocates the idea that autonomic components should be programmed as independent entities with a well defined interaction semantics. This is why the interplay between the programming model and the component architecture is crucial. We believe that bindings should have a uniform semantics ensuring the

independent execution of each component; in GCM/ProActive each binding guarantees the same simple behaviour which is asynchronous requests and replies by futures.

8. CONCLUSION

The design of the GCM component model [1], and its implementation on top of the ProActive parallel and distributed programming library and distributed deployment framework have already started some years ago. Several published articles allowed us to focus on some particular features needed in specific contexts (e.g. collective interfaces for SPMD parallel computing [55], non-functional components for autonomic computing [60, 65], etc). But this paper is original by its own: it presents GCM as an effective programming model, taking the opportunity to expose why and how we have intentionnally taken benefit of a non component-oriented, yet elaborated programming model, the active object model. Consequently, the paper gives a hopefully clear and self-contained view of the way asynchronous and autonomous distributed GCM components run. The reader interested in the formalisation should refer to the work of Henrio et al. [51] for a formal specification of the runtime functional behaviour of GCM/ProActive.

From our recent state of the art analysis of component oriented solutions for distributed programming, it appears that GCM/ProActive stands out for:

- its soundness, given that care has been taken to build upon a well-tried and formalized underlying programming model [76], and that all component oriented specificities and associated operational semantics are formalized in [51];
- its portability upon distributed runtime environments thanks to the Java platform and ProActive open deployment capabilities;
- its genericity and openness regarding all proposed features, both at component model level, and at more practical levels like its interoperability capability with service oriented computing applications through SCA and web services.

So far, numerous GCM applications, from toy ones to larger scope ones, have been developed. For example a message routing layer as a GCM based middleware for federating enterprise service buses or supporting hybrid grid/cloud distributed computing [77], and a peer-to-peer system for web semantics data storage and retrieval through a publish/subscribe approach [78, 79] have been implemented relying on GCM/ProActive.

The perspective here is to benefit not only from highly distributed computing platforms, but also from intra node parallel hardware as multicore and eventually GPU devices, when composing parallel and distributed applications. We are currently working on the replacement of mono-threaded active objects by multi-active objects [59] to implement GCM components: multi-threaded support of GCM method calls can allow GCM asynchronous components to fully benefit from parallel hardware while keeping their loosely-coupled, asynchronous and autonomous nature.

Formalisations on current GCM model and resulting GCM applications behaviour at runtime are also on-going work. In a complementary way, they use theorem proving techniques to formally specify what are the featured properties of the component model, either at the architectural level [80], or concerning its operational semantics [51]; they also model GCM applications regarding the use they make of GCM interfaces, including NF ones, and consequently model-check these models in order to detect if these applications exhibit or not some global properties, like absence of deadlock [81]. Next stage is to succeed to formally prove that GCM autonomous reconfigurations lead the GCM applications into a safe state.

REFERENCES

1. Baude F, Caromel D, Dalmasso C, Danelutto M, Getov V, Henrio L, Prez C. GCM: a grid extension to Fractal for autonomous distributed components. *Annals of Telecommunications* 2009; **64**(1-2):5–24.
2. Bruneton E, Coupaye T, Stefani J. The fractal component model specification 2004. Http://fractal.objectweb.org/specification/index.html.

3. TC-GRID E. ETSI TS 102 827: Grid; grid component model; part 1: Gcm interoperability deployment. *Technical Report*, European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France 2008. URL http://webapp.etsi.org/workprogram/Report_WorkItem.asp?wki_id=26362, standard.
4. TC-GRID E. ETSI TS 102 828: Grid; grid component model; part 2: Gcm application description. *Technical Report*, European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France 2008. URL http://webapp.etsi.org/workprogram/Report_WorkItem.asp?wki_id=30947, standard.
5. TC-GRID E. ETSI TS 102 829: Grid; grid component model; part 3: Gcm fractal architecture description language (adl). *Technical Report*, European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France 2009. URL http://webapp.etsi.org/workprogram/Report_WorkItem.asp?wki_id=28856, standard.
6. TC-GRID E. ETSI TS 102 830: Grid; grid component model; part 4: Gcm fractal java api. *Technical Report*, European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France 2010. URL http://webapp.etsi.org/workprogram/Report_WorkItem.asp?wki_id=28857, standard.
7. Caromel D, Henrio L, Serpette B. Asynchronous and deterministic objects. *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM Press, 2004; 123–134, doi:http://doi.acm.org/10.1145/964001.964012.
8. Service Component Architecture Specifications March 2007. URL http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications.
9. Cca forum homepage. http://www.cca-forum.org/.
10. Bures T, Hnetynka P, Plasil F. Sofa 2.0: Balancing advanced features in a hierarchical component model. *Software Engineering Research, Management and Applications, 2006. Fourth International Conference on*, 2006; 40 –48, doi:10.1109/SERA.2006.62.
11. Seinturier L, Merle P, Rouvoy R, Romero D, Schiavoni V, Stefani JB. A component-based middleware platform for reconfigurable service-oriented architectures. *Software: Practice and Experience* 2012; **42**(5):559–583, doi:10.1002/spe.1077. URL http://dx.doi.org/10.1002/spe.1077.
12. David PC, Ledoux T, Léger M, Coupaye T. Fpath and fscript: Language support for navigation and reliable reconfiguration of fractal architectures. *Annals of Telecommunications* 2009; **64**:45–63. URL http://dx.doi.org/10.1007/s12243-008-0073-y, 10.1007/s12243-008-0073-y.
13. Bertrand, Bramley, B D, Kohl JA, Bernholdt DE, Larson, Sussman A. Data Redistribution and Remote Method Invocation in Parallel Component Architectures. *Proceedings of the 19th International Parallel and Distributed Processing Symposium: IPDPS*, 2005.
14. Prez C, Priol T, Ribes A. A Parallel Corba Component Model for Numerical Code Coupling. *IJHPCA* 2003; **17**(4):417–429.
15. Aldinucci M, Bouziane HL, Danelutto M, Prez C. Stkm on sca: A unified framework with components, workflows and algorithmic skeletons. *Euro-Par 2009 Parallel Processing, 15th International Euro-Par Conference*.
16. Gannon D, Krishnan S, Fang L, Kandaswamy G, Simmhan Y, Slominski A. On building parallel & grid applications: Component technology and distributed services. *Cluster Computing* Oct 2005; **8**(4):271–277, doi:10.1007/s10586-005-4094-2. URL http://dx.doi.org/10.1007/s10586-005-4094-2.
17. Aldinucci M, Danelutto M, Paternesi A, Ravazzolo R, Vanneschi M. Building interoperable grid-aware assist applications via webservices. *Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005,*.
18. Denis A, Prez C, Priol T. Achieving portable and efficient parallel corba objects. *Concurrency and Computation: Practice and Experience* 2003; **15**:891–909.
19. Malawski M, Kurzyniec D, Sunderam VS. MOCCA - Towards a Distributed CCA Framework for Metacomputing.
20. Seinturier L, Pessemier N, Duchien L, Coupaye T. A component model engineered with components and aspects. *Component-Based Software Engineering, 9th International Symposium, CBSE 2006*.
21. Bruneton E, Coupaye T, Leclercq M, Quma V, Stefani JB. The fractal component model and its support in java. *Software: Practice and Experience* 2006; **36**(11-12):1257–1284, doi:10.1002/spe.767.
22. Baumeister H, Hacklinger F, Hennicker R, Knapp A, Wirsing M. A component model for architectural programming. *Electr. Notes Theor. Comput. Sci.* 2006; **160**:75–96.
23. ProActive Parallel Suite. URL http://proactive.inria.fr/.
24. Collet P, Coupaye T, Chang H, Seinturier L, Dufrêne G. Components and services : A marriage of reason. *Technical Report*, CNRS I3S RR 2007.
25. Masek K, Hnetynka P, Bures T. Bridging the component-based and service-oriented worlds. *Software Engineering and Advanced Applications, 2009. SEAA '09. 35th Euromicro Conference on*, 2009; 47 –54, doi:10.1109/SEAA.2009.58.
26. IBM. An architectural blueprint for autonomic computing. *white paper* 2006; **Fourth Edition**(June). URL http://users.encs.concordia.ca/~ac/ac-resources/AC_Blueprint_White_Paper_4th.pdf.
27. David PC, Ledoux T, Léger M, Coupaye T. Fpath and fscript: Language support for navigation and reliable reconfiguration of fractal architectures. *Annals of Telecommunications* 2009; **64**:45–63.
28. Szyperski C. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 2002.
29. Magee J, Kramer J. Dynamic structure in software architectures. *Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, SIGSOFT '96, ACM: New York, NY, USA, 1996; 3–14, doi:10.1145/239098.239104. URL http://doi.acm.org/10.1145/239098.239104.
30. Aldrich J, Chambers C, Notkin D. Archjava: connecting software architecture to implementation. *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, ACM: New York, NY, USA, 2002; 187–197, doi:10.1145/581339.581365. URL http://doi.acm.org/10.1145/581339.581365.
31. Garlan D, Monroe R, Wile D. Acme: Architectural description of component-based systems. *Foundations of component-based systems* 2000; **68**:47–68.

32. Hnětynka P, Plášil F. Dynamic reconfiguration and access to services in hierarchical component models. *Proceedings of the 9th international conference on Component-Based Software Engineering*, CBSE'06, Springer-Verlag, 2006; 352–359, doi:10.1007/11783565_27. URL http://dx.doi.org/10.1007/11783565_27.
33. Broto L, Hagimont D, Stolf P, Depalma N, Temate S. Autonomic management policy specification in tune. *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, ACM: New York, NY, USA, 2008; 1658–1663, doi:10.1145/1363686.1364080. URL http://doi.acm.org/10.1145/1363686.1364080.
34. Coulson G, Blair G, Grace P, Taiani F, Joolia A, Lee K, Ueyama J, Sivaharan T. A generic component model for building systems software. *ACM Trans. Comput. Syst.* Mar 2008; **26**(1):1:1–1:42, doi:10.1145/1328671.1328672. URL http://doi.acm.org/10.1145/1328671.1328672.
35. Tibermacine C, Hoareau D, Kadri R. Enforcing architecture and deployment constraints of distributed component-based software. *Fundamental Approaches to Software Engineering*, *Lecture Notes in Computer Science*, vol. 4422, Dwyer M, Lopes A (eds.). Springer Berlin Heidelberg, 2007; 140–154, doi:10.1007/978-3-540-71289-3_13. URL http://dx.doi.org/10.1007/978-3-540-71289-3_13.
36. Parashar M, Liu H, Li Z, Matossian V, Schmidt C, Zhang G, Hariri S. Automate: Enabling autonomic applications on the grid. *Cluster Computing* 2006; **9**:161–174, doi:10.1007/s10586-006-7561-5. URL http://dx.doi.org/10.1007/s10586-006-7561-5.
37. Joolia A, Batista T, Coulson G, Gomes ATA. Mapping adl specifications to an efficient and reconfigurable runtime component platform. *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*, WICSA '05, IEEE Computer Society: Washington, DC, USA, 2005; 131–140, doi:10.1109/WICSA.2005.42. URL http://dx.doi.org/10.1109/WICSA.2005.42.
38. David PC, Ledoux T. An aspect-oriented approach for developing self-adaptive fractal components. *Software Composition*, *Lecture Notes in Computer Science*, vol. 4089, Löwe W, Südholt M (eds.). Springer Berlin / Heidelberg, 2006; 82–97. URL http://dx.doi.org/10.1007/11821946_6, 10.1007/11821946_6.
39. Buisson J, Andr F, Pazat JL. A framework for dynamic adaptation of parallel components. *Parallel Computing: Current & Future Issues of High-End Computing International Conference ParCo*, *NIC Series*, vol. 33, Malaga Spain, 2005; 65. URL http://hal.archives-ouvertes.fr/hal-00498836/en/.
40. Garlan D, Cheng SW, Huang AC, Schmerl B, Steenkiste P. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer* 2004; **37**:46–54.
41. Danelutto M, Kilpatrick P, Montangero C, Semini L. Model checking support for conflict resolution in multiple non-functional concern management. *Euro-Par 2011: Parallel Processing Workshops*, *Lecture Notes in Computer Science*, vol. 7155. Springer Berlin Heidelberg, 2012; 128–138, doi:10.1007/978-3-642-29737-3_16. URL http://dx.doi.org/10.1007/978-3-642-29737-3_16.
42. Gueye S, Palma N, Rutten E. Component-based autonomic managers for coordination control. *Coordination Models and Languages*, *Lecture Notes in Computer Science*, vol. 7890, Nicola R, Julien C (eds.). Springer Berlin Heidelberg, 2013; 75–89, doi:10.1007/978-3-642-38493-6_6. URL http://dx.doi.org/10.1007/978-3-642-38493-6_6.
43. Barros T, Cansado A, Madelaine E, Rivera M. Model-checking distributed components: The vercors platform. *Electronic Notes in Theoretical Computer Science* 2007; **182**(0):3 – 16, doi:10.1016/j.entcs.2006.09.028. URL http://www.sciencedirect.com/science/article/pii/S1571066107003829, proceedings of the Third International Workshop on Formal Aspects of Component Software (FACS 2006).
44. Moreira R, Blair S, Carrapatoso E. Supporting adaptable distributed systems with formaware. *Distributed Computing Systems Workshops, 2004. Proceedings. 24th International Conference on*, 2004; 320 – 325, doi:10.1109/ICDCSW.2004.1284049.
45. Lger M, Ledoux T, Coupaye T. Reliable dynamic reconfigurations in a reflective component model. *Component-Based Software Engineering*, *Lecture Notes in Computer Science*, vol. 6092, Grunske L, Reussner R, Plasil F (eds.). Springer Berlin Heidelberg, 2010; 74–92, doi:10.1007/978-3-642-13238-4_5. URL http://dx.doi.org/10.1007/978-3-642-13238-4_5.
46. Monroe R. Capturing software architecture design expertise with armani. *Technical Report CMU-CS-98-163*, Carnegie Mellon University School of Computer Science January 2001. Version 2.3.
47. Wang N, Schmidt DC, OŔyan C. *An Overview of the CORBA Component Model*. Addison-Wesley, Reading, 2001.
48. Sicard S, Boyer F, De Palma N. Using components for architecture-based management: the self-repair case. *Proceedings of the 30th international conference on Software engineering*, ICSE '08, ACM: New York, NY, USA, 2008; 101–110, doi:10.1145/1368088.1368103. URL http://doi.acm.org/10.1145/1368088.1368103.
49. Johnsen EB, Owe O, Yu IC. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science* Nov 2006; **365**(1–2):23–66.
50. Lienhardt M, Lanese I, Bravetti M, Sangiorgi D, Zavattaro G, Welsch Y, Schäfer J, Poetzsch-Heffter A. A component model for the abs language. *Proceedings of the 9th international conference on Formal Methods for Components and Objects*, FMCO'10, Springer-Verlag, 2011; 165–183, doi:10.1007/978-3-642-25271-6_9. URL http://dx.doi.org/10.1007/978-3-642-25271-6_9.
51. Henrio L, Kammller F, Rivera M. An asynchronous distributed component model and its semantics. *FMCO 2008*, *Lecture Notes in Computer Science*, vol. 5751, de Boer FS, Bonsangue MM, Madelaine E (eds.), Springer, 2009; 159–179.
52. Lavender RG, Schmidt DC. Active object: an object behavioral pattern for concurrent programming. *Pattern languages of program design 2*. Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 1996; 483–499.
53. Agha G. *Actors: a model of concurrent computation in distributed systems*. MIT Press: Cambridge, MA, USA, 1986.
54. Halstead, Jr RH. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 1985; **7**(4):501–538, doi:http://doi.acm.org/10.1145/4472.4478.

55. Mathias E, Cavé V, Lanteri S, Baude F. Grid-enabling SPMD Applications through Hierarchical Partitioning and a Component-Based Runtime. *15th Int. European Conference on Parallel and Distributed Computing (Euro-Par 2009)*, *LNCS*, vol. 5704, 2009; 691–703.
56. Baduel L, Baude F, Caromel D, Contes A, Huet F, Morel M, Quilici R. *Grid Computing: Software Environments and Tools*, chap. Programming, Deploying, Composing, for the Grid. Springer-Verlag, 2006. ISBN 978-1-85233-998-2.
57. Charron-Bost B, Mattern F, Tel G. Synchronous, asynchronous, and causally ordered communication. *Distributed Computing* 1996; **9**:173–191. URL http://dx.doi.org/10.1007/s004460050018, 10.1007/s004460050018.
58. Cansado A, Henrio L, Madelaine E. Transparent first-class futures and distributed component. *International Workshop on Formal Aspects of Component Software (FACS'08)*, Electronic Notes in Theoretical Computer Science (ENTCS): Malaga, 2008.
59. Henrio L, Huet F, Istvn Z. Multi-threaded active objects. *COORDINATION 2013*, Julien C, De Nicola R (eds.), LNCS, IFIP International Federation for Information Processing, Springer, 2013. 15th International Conference on Coordination Models and Languages, Florence, Italy, 3–6.
60. Baude F, Caromel D, Henrio L, Naoumenko P. A flexible model and implementation of component controllers. *Making Grids Work*. Springer US, 2008; 31–43, doi:10.1007/978-0-387-78448-9_3. URL http://dx.doi.org/10.1007/978-0-387-78448-9_3.
61. Henrio L, Rivera M. Stopping safely hierarchical distributed components. *Proceedings of the Workshop on Component-Based High Performance Computing (CBHPC'08) in conjunction with ACM SIGPLAN CompArch 2008*, 2008.
62. Horn P. Autonomic computing: Ibm's perspective on the state of information technology. *Computing Systems* 2001; **2007**(Jan):1–40. URL http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf.
63. Kephart J, Chess D. The vision of autonomic computing. *Computer* Jan 2003; **36**(1):41 – 50, doi:10.1109/MC.2003.1160055.
64. SCA Policy Framework Service Component Architecture Specifications Dec 2011.
65. Ruz C, Baude F, Sauvan B. Using Components to Provide a Flexible Adaptation Loop to Component-based SOA Applications. *International Journal on Advances in Intelligent Systems* Jul 2012; **5**(1 & 2):32–50. URL www.iariajournals.org/intelligent_systems/, iSSN: 1942-2679.
66. Aldinucci M, Danelutto M, Kilpatrick P. Autonomic management of non-functional concerns in distributed and parallel application programming. *Proc. of Intl. Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2009.
67. Gauvrit G, Daubert E, Andr F. SAFDIS: A Framework to Bring Self-Adaptability to Service-Based Distributed Applications. *36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, 2010; 211–218.
68. Fouquet F, Daubert E, Plouzeau N, Barais O, Bourcier J, Jzquel J. Dissemination of reconfiguration policies on mesh networks. *DAIS*, *Lecture Notes in Computer Science*, vol. 7272, 2012; 16–30.
69. Bures T, Gerostathopoulos I, Hnetynka P, Keznikl J, Kit M, Plasil F. Deeco: an ensemble-based component system. *CBSE*, ACM, 2013; 81–90.
70. Ghosh D, Sheehy J, Thorup K, Vinoski S. Programming language impact on the development of distributed systems. *Journal of Internet Services and Applications* 2012; **3**(1):23–30, doi:10.1007/s13174-011-0042-y. URL http://dx.doi.org/10.1007/s13174-011-0042-y.
71. Haller P, Odersky M. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science* 2009; **410**(2-3):202–220.
72. Gupta M. *Akka Essentials*. Packt Publishing Ltd, 2012.
73. Hähnle R, Helvensteijn M, Johnsen E, Lienhardt M, Sangiorgi D, Schaefer I, Wong P. Hats abstract behavioral specification: The architectural view. *Formal Methods for Components and Objects*, *LNCS*, vol. 7542. Springer, 2013; 109–132, doi:10.1007/978-3-642-35887-6_6. URL http://dx.doi.org/10.1007/978-3-642-35887-6_6.
74. Schäfer J, Poetzsch-Heffter A. JCoBox: Generalizing active objects to concurrent components. *ECOOP 2010 – Object-Oriented Programming* 2010; :275–299.
75. Bures T, Malohlava M, Hnetynka P. Using dsl for automatic generation of software connectors. *Composition-Based Software Systems, 2008. ICCBSS 2008. Seventh International Conference on*, 2008; 138–147.
76. Henrio L, Kammller F, Lutz B. ASPfun : A typed functional active object calculus. *Science of Computer Programming* 2012; **77**:823–847, doi:DOI:10.1016/j.scico.2010.12.008. URL http://www.sciencedirect.com/science/article/B6V17-51XR3D4-1/2/52153869e8c4df60a65b2d52bbf9828c.
77. Mathias E, Baude F. A Component-Based Middleware for Hybrid Grid/Cloud Computing Platforms. *Concurrency and Computation: Practice and Experience* 2012; **24**(13):1461–1477.
78. Filali I, Pellegrino L, Bongiovanni F, Huet F, Baude F. Modular P2P-Based Approach for RDF Data Storage and Retrieval. *AP2PS 2011, The Third International Conference on Advances in P2P Systems*, 2011; 39–46. Best paper award.
79. Baude F, Bongiovanni F, Pellegrino L, Quéma V. Requirement Specification for the Event Cloud Component, Deliverable D.2.1. *Technical Report*, FP7 Play EU project Sept 2011. www.play-project.eu.
80. Nuno G, Henrio L, Madelaine E. Bringing coq into the world of gcm distributed applications. *International Journal of Parallel Programming - Special issue HLPP'2013* 2013; To appear.
81. Boulifa RA, Halalai R, Henrio L, Madelaine E. Verifying safety of fault-tolerant distributed components. *International Symposium on Formal Aspects of Component Software (FACS 2011)*, Lecture Notes in Computer Science, Springer, 2011.